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@univie.ac.at

***Abstract*—The Domain Name System (DNS) is a core component of the Internet. Clients query DNS servers to translate domain names to IP addresses. Local DNS caches alleviate the time it takes to query a DNS server, thereby 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, system-wide 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 82.86 % and 78.89 % for detecting accesses to 10 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 [68], [69]. 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 [68]. DNS caches alleviate the latency problem and reduce traffic by storing recently resolved domain names, associated IP addresses, and their corresponding time-to-live (TTL). Later requests for the same domain are answered immediately from the cache.

Caches introduce side channels [78], allowing an adversary to probe whether specific data has been previously accessed. Since DNS caches may be shared on a local machine, in a router, or any part of the DNS resolution chain, an attacker can exploit the cache [26] to spy on users' (potentially sensitive) web activity. DNS cache timing attacks [26], [32], [67] exploit timing differences between hits and misses to infer visited domains. Grangeia [32] summarized three methods to measure the state of a network DNS cache: (1) preventing recursive resolution, (2) setting of a low TTL, and (3) measuring the domain resolution latency. Reading from the cache is a destructive operation, *i.e.*, the cache state is modified by reading and has to be reset before it can be exploited again. Felten and Schneider [26] were the first to discuss information leakage from timings of the local DNS cache. Their work, more than 2 decades old, only performs a limited experiment with 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 evaluate success rates and reliability in a continuous monitoring scenario. Concurrently with our work, Moav et al. [67] proposed a similar DNS cache timing attack, targeting the router's DNS cache instead of the local system-wide DNS cache, allowing them to monitor IoT device behavior as well. In contrast, we focus on `systemd-resolved`, a local caching DNS resolver used in popular Linux distributions [16]. While flushing the DNS cache in `resolved` is an unprivileged operation, the corresponding `resolvectl` interface is not available in VMs, application sandboxes, and web browsers. We demonstrate reliable eviction from these restricted environments, which is a prerequisite for an Evict+Reload-style attack [35] on the local DNS cache.

In this paper, we introduce DNS Monitoring via Timing (DMT), a novel Evict+Reload attack on the victim's **local** DNS cache from unprivileged, sandboxed contexts. DMT continuously monitors a victim's activity by combining known

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.242287
www.ndss-symposium.org

timing primitives to probe the DNS cache with novel **reliable DNS cache eviction** techniques: We present 4 techniques to evict the local DNS cache from 3 restricted environments, including VMs, and websites with and without JavaScript. We also introduce 3 measurement primitives, enabling us to measure DNS resolution timing from native code, JavaScript, and scriptless HTML, respectively. Combining 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. Our eviction techniques achieve a similar reliability as `resolvectl flush-caches`, which is not available in restricted environments. One of our eviction primitives exploits specific behavior of `resolved`, whereas the other 3 are applicable to other resolvers as well.

DMT can be mounted remotely by an off-path attacker, via JavaScript code or even plain HTML. We perform only a single timing measurement to determine whether a distinct domain was accessed. Since we do not rely on transmitted content, data, or any other website characteristics, DMT does not qualify as a fingerprinting attack. Changes to the target website's content do not affect the attack's reliability. In fact, when mounted from a browser, DMT benefits from strict *Cross Origin Resource Sharing* (CORS) policies, as they prevent the browser from loading the target site's contents, reducing noise in our measurements. When operating from native code, no connection is established to the target server at all. Altogether, this makes DMT less susceptible to misclassification compared to classic website fingerprinting attacks [38].

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 a 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 nearly all cases when performing the attack from native code using system-provided DNS APIs. In the absence of direct access to system-wide APIs, DMT leverages other, widely available APIs (e.g., JavaScript `fetch`), triggering implicit domain resolutions. In these experiments, we achieved a worst-case false-negative rate of 24.2 % per measurement. Execution times vary depending on the scenario: In the best case, the cross-VM attack, our attack takes, on average, 763.122 ms to monitor 100 domains, and each additional domain takes another 2.931 ms. In the worst case, the JavaScript-based attack, it takes 14.787 s for 100 domains, and an increase of 147.110 ms per additional domain.

Our unprivileged native code end-to-end experiment *within a VM* detects accessed domains by the host from a list of 103 websites (open world). We achieved an F_1 score of 92.48 % in 685.855 ms ($n = 3000$, $\sigma_{\bar{x}} = 14.893$ ms). If the DNS server returns errors, subsequent eviction takes either 77.267 ms ($n = 2562$, $\sigma_{\bar{x}} = 0.287$ ms) (in case the DNS server reports errors), and 5 s if it times out. 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 the reliability of our attack, practically halving the false-negative rate in most experiments, and achieving a worst-case false-negative rate of 13.6 %. Finally, we discuss that DMT works within a VPN setting.

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 VMs), 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.
- In the best case, the cross-VM attack, monitoring 100 domains takes 763.122 ms with an increase by 2.931 ms for any additional domain. In the worst case, the JavaScript-based attack, it takes 14.787 s and each additional domain adds another 147.110 ms.
- Our end-to-end attack from inside a VM reliably detects host website visits from a list of 103 websites (open-world) with an F_1 score of 92.48 % in less than one second.

Outline. We discuss concurrent work on DNS cache timing attacks in Section II, and background in Section III. Section IV provides an overview of our attack. We present techniques to read and evict the DNS cache in Sections V and VI. Section VII evaluates our cross-VM attack and Section VIII our website access tracking from JavaScript in the browser. We discuss and conclude our work in Sections IX and X.

II. Concurrent Work

Recent work by Moav et al. [67] was not available at the time of this paper's original submission, but is public as of August 2025. They describe a similar DNS cache timing attack to track user- and IoT device activity. In contrast to our paper, which focuses on the local OS DNS cache, they focus on the DNS forwarder in the router as the attack target. While they also evaluate systems with `systemd-resolved`, their work considers evicting the local OS cache to force DNS requests to propagate to the router's DNS resolver. Summarizing, while the attacks are similar in nature, they target different caches in the DNS resolution chain. This is also highlighted by the fact that they can attack IoT devices, which is not possible with our attack. However, our experiments show a more reliable side-channel measurement due to significantly lower latencies for cache hits. Furthermore, DMT can be mounted even if the system is configured to use a common public DNS server or a VPN, since the local DNS cache leaks information irrespective of the used upstream DNS server and network path. Thus, the two attacks are orthogonal but complementary. For transparency, Table VI in the Appendix lists experiments performed after *DNS FLaRE* was published.

III. 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) translates human-readable domain names into IP addresses [68], [69], and is a distributed infrastructure: The root name servers resolve the top-level domains (TLDs), such as `.com` and `.net`. Next, the TLDs have ad-hoc name servers for the resolution of second-level domains (SLDs). These then point to the authoritative servers that know the records for specific domains and are usually 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 may 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 [24].

Over time, more information has been incorporated into DNS, e.g., SPF [48], DKIM [19] or DMARC [51]. DNS also plays a role in malware protection [85], parental control systems [57], censorship [39], and DDoS protection [46]. Due to its importance, DDoS attacks against the DNS have serious consequences for the Internet as a whole [94].

Traditional DNS is neither encrypted nor integrity-protected, facilitating cache poising [62], interception [83], or user fingerprinting [6], which has led to numerous improvements in domain resolution. Domain Name System Security Extensions (DNSSEC) [40] provides cryptographic authentication of DNS resource records, but does not encrypt the queries and replies. This changed with DNS over HTTPS (DoH) [41] and DNS over TLS (DoT) [42] tunneling DNS requests over 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, causing a non-negligible delay of up to hundreds of milliseconds [45]. Repeated resolutions of the same domain name are 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 [68], [69]. The TTL defines a resource record's lifetime in seconds and therefore limits the time that a response is cached. At a later point in time, negative caching has been introduced by DNS [4], [99], *i.e.*, negative results are also stored in the cache. Recommendations for the TTL vary between five minutes and 24 hours, depending on the distinct scenario, introducing a tradeoff between performance and flexibility [70].

In practice, a multi-level caching architecture for DNS emerged, frequently reducing DNS requests to a round-trip time of a few milliseconds [15]. Recursive servers 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 provide DNS caches that are shared by all their applications. When a resource is cached, no communication over the network is necessary for DNS resolution. Even browsers operate their own caches, limiting coordination with the operating system. Modern Chromium- and Firefox-based browsers also implement DNS-over-HTTPS (DoH) resolvers, bypassing the OS-provided DNS cache entirely in some configurations [17], [71].

Caches have also 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 [62], [63], [56] lead to the storage of illegitimately modified resources. Upon request, these poisoned records are forwarded to the clients, tricking them into a connection with a potentially malicious destination. Third, the differences in timing between cached and uncached records form a timing side channel [26].

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 [50]. Timing can originate in software [50] or hardware, e.g., due to caching [77], [100]. For cache-timing side channels, there are generic techniques like Flush+Reload [100], Evict+Reload [35], Prime+Probe [77], [59], [64], and Flush+Flush [34]. 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* [26], [76], [33], [97]. Special APIs can be attacked remotely [10], [13], [18], [87], [20]. Some remote attacks even only send packages to a victim and observe timing differences through this [53], [29].

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 [5], [73], [93], [88]. 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 [95], [90], videos [25], [89], and websites [84], [11]. Early work by Hintz [38] 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 [12], packet direction ratios, and total packet counts [80], [79]. Klein and Pinkas [49] 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 [14] 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 [37], [98], [36], [84], [11], [91], [92], [88], [9], [22], [44]. Additionally, network side channels have been demonstrated remotely, both over the Tor network [74], [65] and on standard network infrastructure [47], [30], [3], [29], with more active techniques requiring interaction between the victim and a remote server.

Most closely related to our work are DNS cache timing attacks [26], [32], [81], [67] and DNS cache flushing attacks [1]. DNS cache timing attacks infer a resource's presence in a cache. Felten and Schneider [26] were the first to demonstrate that DNS cache timings can be used as a side channel to infer whether a domain name is cached by the DNS resolver. Grangeia [32] 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 [26]. Klein and Pinkas [49] exploited DNS cache timing differences to track users and pointed 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. Afek et al. [1] demonstrated that DNS caches can be flushed remotely by sending specific DNS queries to resolvers, allowing an attacker to perform a denial-of-service on a public shared DNS resolver. Moav et al. [67] demonstrate a flush-reload attack against DNS forwarders in routers to track users and IoT devices. Shared and public DNS resolvers can be targeted with these methods to investigate domain access distribution for user access statistics [81], [43], [55], [75], [54], [82], user tracking [58], and malicious domain detection [61], [27], [28], [60]. Beyond that, public resolver caches were exploited for covert communication channels [86]. In practice, however, success depends on the resolver's specific cache structure [66]. A single shared cache for all clients has the highest hit rates, whereas isolation incurs high overheads. Some recursive resolvers do not cache at all, potentially for security reasons.

IV. High-Level Overview and Threat Model

In this section, we provide a threat model and high-level overview for the DMT 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, system-wide 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 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 like 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 scam's perceived credibility 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.

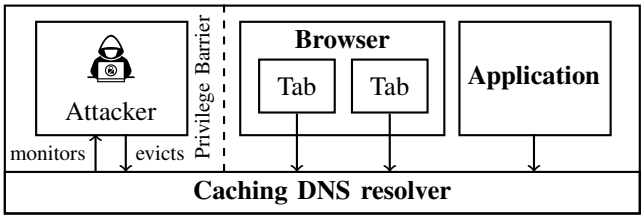

Fig. 1. General overview of the working principle of DMT. Applications (e.g., browsers) interact with the DNS resolver. Resolved domain names are added to the shared system-wide DNS cache. The unprivileged attacker monitors the state of the DNS cache, which leaks information about user behavior and running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

We assume privacy-concerned victims may use a VPN or 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 local DNS cache by measuring the resolution latency of target domain names. If the domain name is not cached, the system-wide DNS resolver needs to perform a network round-trip to an external DNS server. This round-trip introduces significant delay to the resolution process. The measurement is destructive, as the target domain is added to the cache. Thus, the attacker needs to evict the DNS cache after each measurement to improve the temporal resolution of the attack. Therefore, we introduce multiple DNS cache eviction primitives and compare them to the cache-flushing commands available only in native, non-sandboxed code. These primitives allow an

attacker to evict the DNS cache from application sandboxes, VMs, 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 V) and to evict the DNS cache (as we discuss in Section VI). 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 the browsing activities of their coworkers. Lightweight application sandboxes, such as Firejail and Docker, as well as VMs in a NAT networking setup, can rely on the system-wide DNS resolver, leaving our attacks unaffected. We even find this to be the default on a Debian 12 installation in a libvirt VM, resulting in the same capabilities as a regular non-privileged user on the host, *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 VI).

Remote JavaScript. Since our measurement and eviction primitives only require DNS resolution (cf. Section VI), 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 on a third-party website, e.g., via ad networks or vulnerabilities on 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. We can then measure the timing difference between these two requests 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.

V. Measuring the Local DNS Cache State

In this section, we analyze timing differences between cached and uncached domain names in different scenarios, including JavaScript code and plain HTML in the browser, as well as native code. We focus on `systemd-resolved`, the default resolver in e.g., Fedora and Ubuntu [16], and recommended caching DNS resolver on Arch Linux [7]. Distributions not using `systemd-resolved` typically do not ship with a system-wide DNS cache by default. Even though our evaluation focuses on `systemd-resolved`, the general principles of our attack apply to any system-wide DNS cache (e.g., Windows or macOS), as caching inherently introduces timing differences.

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. We experimentally confirmed that we can achieve similar results using Firefox 144.0.2 (cf. Figure 12). 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 can execute `resolvectl query`, they can see whether the resource was cached directly, eliminating the need for timing measurements. In native code contexts without access to `resolvectl`, the attacker can use native library functions to resolve a domain name and measure the latency of the function call. However, 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 HTTP request, its latency is influenced by the web server's response time. In a scriptless environment, the attacker relies on requests to attacker-

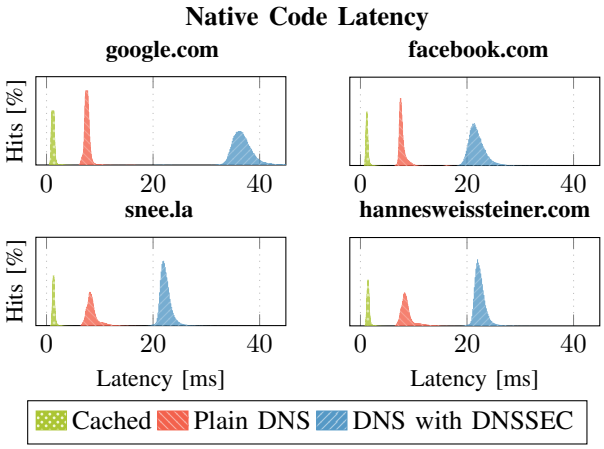

Fig. 2. Histogram of domain resolution latencies for cached and uncached domain names, with and without DNSSEC enabled, recorded in native code. There is a clear separation between cached and uncached domain names. With DNSSEC enabled, the latency difference is even larger.

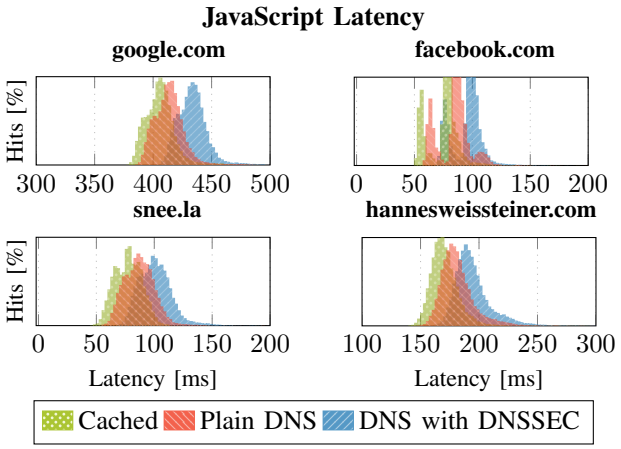

Fig. 3. Latency histogram for DNS resolutions using `fetch` on JavaScript. Compared to Figure 2, the measurements are significantly noisier, but still allow us to distinguish between cached and uncached domain names. The latency differences between websites are caused by their server response times.

controlled servers before and after target requests to measure their runtime, adding even more noise due to the two additional network requests. In the following, we describe and evaluate measurement strategies for *native code*, *JavaScript* and *scriptless HTML*. Additionally, we also evaluate the JavaScript experiments on two different consumer Internet connections, demonstrating the impact of Internet speeds on our attack.

A. Native Code

On Linux, the attacker cannot directly view the DNS cache, as `resolvectl show-cache` is a privileged operation (in contrast, the equivalent PowerShell command on Windows is unprivileged, allowing for cross-user leakage). Yet, attackers with shell access can use the unprivileged `resolvectl query` for resolution, which directly reports whether the response was cached. If `resolvectl` is not available (e.g., in a jailed environment or a VM), the attacker can instead measure the resolution latency to distinguish cached from uncached domain names, using programs like `dig` or functions like `libc`'s `getaddrinfo` or Python's `socket.gethostbyname` to resolve domain names.

Evaluation. We measure the execution time of Python's `socket.gethostbyname`. 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, taking around 20 ms, due to signature validation for each response.

B. JavaScript

In the browser, there 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 a longer runtime. However, as `fetch` performs a full HTTP request, the web server latency adds measurement noise.

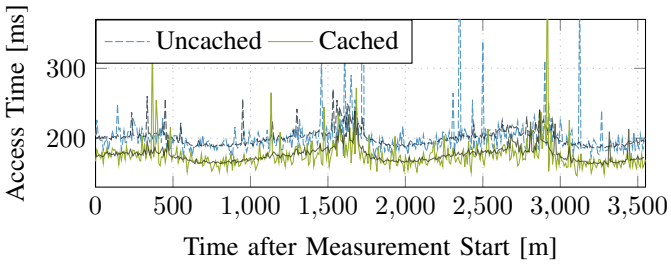

Fig. 4. Trace of `fetch` latencies for cached and uncached accesses of `snee.la` over approximately 60 h. Bright lines show one raw sample every 500 seconds, while dark lines show the average of 50 samples during the same timeframe. The offset between cached and uncached latencies stays relatively constant over time, while the absolute latencies vary, adding noise to long-term measurements, while not affecting actual cache state measurements.

Cross-Origin Resource Sharing (CORS) requires the browser to send a preflight `OPTIONS` request in advance of the actual request, to check CORS HTTP headers without loading any content. As most websites forbid cross-origin requests, the `fetch` request will fail directly after the preflight response. While the `OPTIONS` request still adds noise to the measurement (compared to only resolving the domain), the website's content does not have an impact as it is not loaded.

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 Figure 3 and Figure 5 show a peculiar histogram shape for

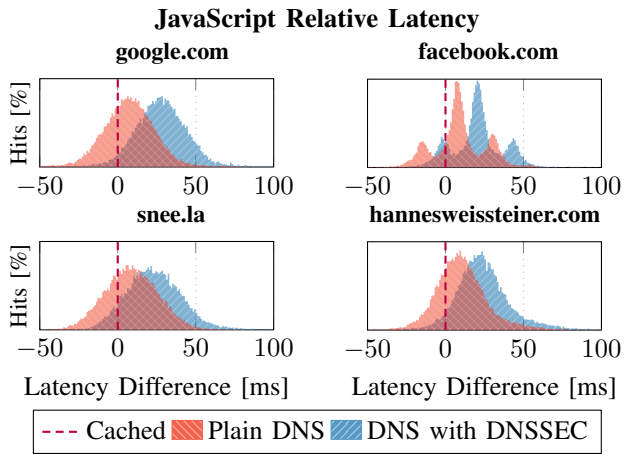

Fig. 5. Histogram of the latency differences between an uncached and the following cached domain lookup, when using `fetch` in JavaScript. This representation reduces the noise caused by slow changes in the server response time, as seen in Figure 4. This visualization better highlights the measurable latency difference between cached and uncached accesses.

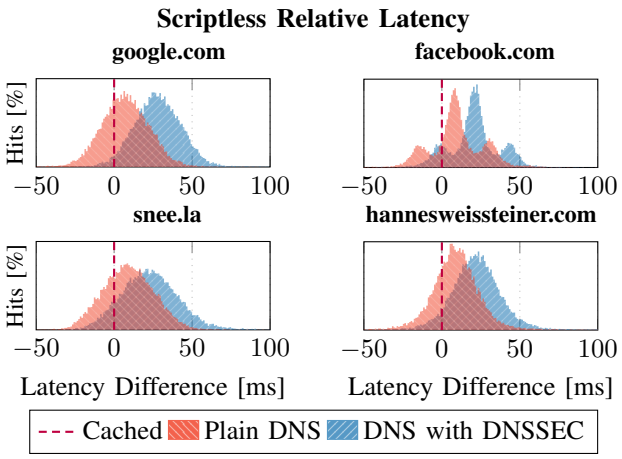

Fig. 6. Latency difference histogram with a scriptless measurement. The latency differences are similar to the JavaScript measurements in Figure 5, indicating that our method of using alternative fonts to serialize website requests does not introduce significant noise.

```
1  body {
2    font-family: "DMTFont";
3  }
4
5  @font-face {
6    font-family: "DMTFont";
7    src: url("https://attacker.com/measurement-start"),
8         url("https://target-domain.com/random-value"),
9         url("https://attacker.com/measurement-end");
10 }
```

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.

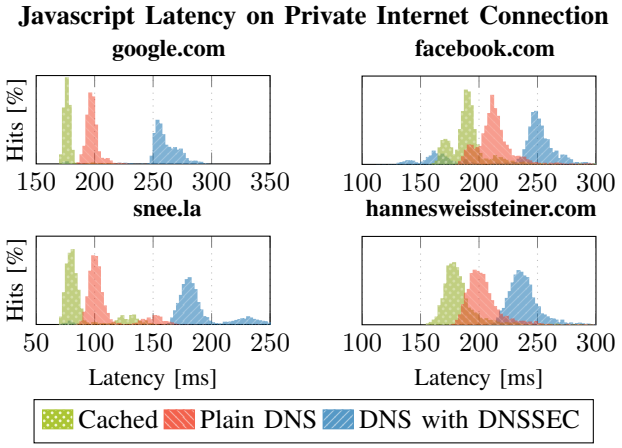

Fig. 7. Latency histogram on a 50 Mbit/s private Internet connection. In contrast to Figure 3, the distributions are significantly more separated. This demonstrates that our other measurements, recorded on a datacenter-grade connection, show the worst-case scenario for our attack.

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

When JavaScript is unavailable (e.g., for security reasons), DMT can be mounted from a plain HTML page. In our scriptless attack, the attacker exploits CSS features [96], in particular the font fallback system. Fallback fonts are loaded sequentially; we exploit them to infer the latency of 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. Similarly, the font fallback system can also be used to evict the browser's DNS cache, as described in Section VI-E.

Evaluation. For a realistic measurement, we measure latencies across 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 effects.

D. Consumer Internet Connection

All previous experiments were performed on a VPS in a commercial data center with a fast Internet connection (download: 1 960 Mbit/s, upload: 1 518 Mbit/s, latency to Google DNS: 5 ms). Additionally, we performed the worst-performing experiment, the JavaScript-based attack, on a low-end private Internet connection (ADSL, download: 48 Mbit/s, upload: 8.4 Mbit/s, latency: 13.5 ms), as well as a mid-tier cable connection (download: 299.9 Mbit/s, upload: 52.2 Mbit/s, latency:

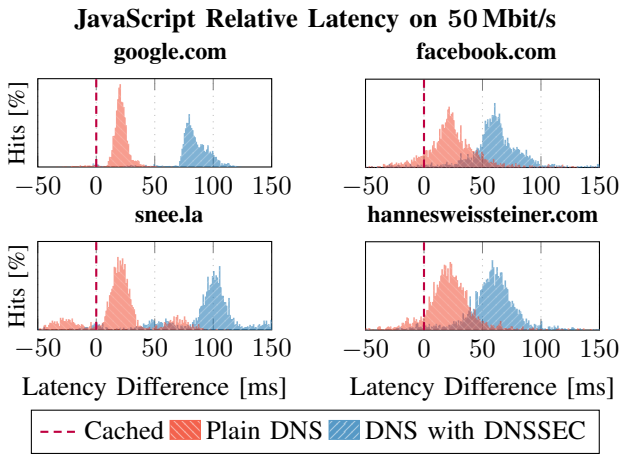

Fig. 8. Latency difference histogram of JavaScript on the 50 Mbit/s connection. Compared to Figure 5, which was measured on a cloud VPS, we experience larger latency differences, resulting more reliable measurements.

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 %
JS 50 Mbit/s[2]	✓	82.363 ms	18.041 ms	1.110 %
	✗	22.902 ms	31.044 ms	13.209 %
JS 300 Mbit/s[3]	✓	35.998 ms	35.368 ms	9.839 %
	✗	16.567 ms	34.029 ms	19.630 %
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 (likely due to connectivity issues), we eliminated the top and bottom 0.1 % 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] Private 50 Mbit/s connection, approximately 2 500 samples.
[3] Private 300 Mbit/s connection, approximately 5 000 samples.

10.7 ms). Figure 8 shows the latency difference histogram for JavaScript on the low-end connection. 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. Relative histograms for the 300 Mbit/s connection can be found in Appendix C, along with a measurement on Firefox (Figure 12). The results highlight that a low-latency, high-bandwidth connection is the worst-case scenario from an attacker's perspective. Table I summarizes the average latency differences, standard deviations, and false negative rates for all our experiments on **facebook.com**, for the VPS as well as both private Internet connections.

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 to Google's DNS server. With shorter latencies, the separation between cached and uncached domains becomes more difficult. Yet, there is a trend towards public resolvers, which often have features like DNS-over-TLS, increased privacy, or reduced filtering.

Application-internal DNS Resolvers. Because our attack relies on measuring the contents of the system-wide DNS cache, it is ineffective if applications bypass the system-wide DNS resolver entirely. For instance, Chromium-based browsers implement their own DNS resolver with an integrated cache [17]. Firefox offers an option to resolve domain names via DNS-over-HTTPS (DoH) [71], also bypassing the OS resolver. Without access to the system-wide cache, DMT cannot monitor its contents. However, both browsers still access the system-wide DNS cache in common configurations. On Linux systems, `systemd-resolved` acts as a locally-running DNS server, causing Chromium-based browsers to connect to it by default, even when DoH is enabled, reenabling the attack. However, when using the *Default Protection* mode, DoH is turned off in all except 4 countries worldwide [71], leaving Firefox users in most countries vulnerable to DMT by default, on all operating systems. Even when configured to use DoH, browsers still fall back to the system-wide DNS resolver if a domain cannot be resolved via DoH [17], [71]. This fallback mechanism is necessary to resolve network-local domain names, e.g., in corporate networks.

Network State Partitioning. Both Chromium and Firefox implement network partitioning, which separates various caches, including the DNS cache, based on the origin of the loaded website [31], mitigating cache-based information leakage across different websites. However, DMT can still monitor the system-wide DNS cache by evicting the attacker website's browser DNS cache, which forces the browser to access the system-wide DNS cache. Still, network state partitioning leads to fewer DNS requests from other tabs hitting the system-wide DNS cache, reducing the attack's accuracy.

HTTPS DNS Records. HTTPS DNS records are used to advertise a domain's HTTPS configuration, improving security and connection speed. Before making an HTTP request, the browser first resolves the domain's HTTPS DNS record, adding additional latency to the DNS resolution. Moav et al. [67] avoid this extra latency by prefetching the HTTPS DNS records of target domains. This is required for their attack, as they measure the DNS latency by accessing port 0, which uses a different set of HTTPS records than a regular website access. However, our browser-based measurements use regular HTTPS DNS records, which can be cached. Thus, the extra latency of resolving HTTPS DNS records during a cache miss increases the latency difference between cached and uncached domains, even improving our attack's accuracy.

F. Summary

Table I summarizes the results of our measurement primitives. 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 negatives. From the browser, we can only resolve domain names using a full web request, adding noise to our measurements. For most websites, CORS preflight requests minimize this noise by terminating the request before any data is downloaded. Still, we measure a false-negative rate of 24.192 % in JavaScript, and 23.929 % in plain HTML, allowing us to distinguish cached and uncached resolutions with a high level of confidence in few measurements. DNSSEC increases the DNS resolution 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.

VI. 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. Most browsers implement an additional DNS cache, which, in principle, is also vulnerable to timing attacks. However, as this cache is not shared with the rest of the system and implements features like network state partitioning [31], it is not possible to monitor other applications or websites using the browser DNS cache. Thus, we also demonstrate how we bypass this cache in Chromium and Firefox.

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. To force the victim to access our DNS server, we set up `NS` records for a domain we control, pointing to our server's IP address. This delegates the responsibility of resolving the subdomains to our malicious DNS server. We use random subdomain prefixes to bypass server- and client-side DNS caches for requests to our DNS server. The DNS server selected by the victim client (e.g., Google DNS or ISP-default) recursively resolves the subdomain and forwards the result to the victim, giving us control over DNS responses.

A. Direct Cache Flushing

The **first** primitive is a direct cache flush. On many common caching DNS resolvers, such as Windows, or `systemd-resolved` on Linux, 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 shell access on the system. Thus, from a sandboxed environment, such as FireJail or a VM, this primitive is not available.

B. Individual DNS Requests

The **second** primitive evicts the cache by filling it with random individual entries. Even though RFC 1536 recommends that DNS cache sizes should be unbounded [21], common real-world DNS caches (e.g., `systemd-resolved`, Android's `resolv`) have a fixed maximum cache size, restricting memory usage. Additionally, many 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 from all execution contexts. With a resolution time of approximately 30 ms per domain and sequential resolutions, the eviction takes over 2 min for `systemd-resolved`, with a cache size of 4 096 entries. Subsequent evictions are faster, as they only need to remove entries that have been evicted by legitimate DNS requests. The attacker can significantly reduce eviction time by parallelizing DNS requests. In our experiments, parallel resolution evicted the entire cache in 5.109 s using 100 threads. As this primitive fills the cache with long-TTL entries, subsequent legitimate DNS requests evict each other. Thus, this primitive 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 of 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, eliminating the single-domain limitation of the previous primitive.

Other operating systems. We also tested this eviction primitive on macOS Sonoma (14.7.4), as well as Windows 11. Both operating systems are closed-source, and we did not find any documentation on their DNS cache eviction strategies. On macOS, we found that we could evict legitimate entries by filling the cache with random requests. This allows a similar evict-and-reload attack on macOS. However, eviction is less reliable than on `systemd-resolved`. Appendix B describes the experiment in more detail. On Windows 11, we were not able to find a working eviction strategy without shell access. The `Get-DnsServerCache` command reported up to 160 000 entries in the cache during our tests, indicating that Windows does not have a fixed-size cache. While the

cache occasionally evicts old entries, we did not find a way to trigger this cleanup process reliably However, we observed a clear timing difference between cached and uncached domains. Concluding, while there is no known eviction strategy yet, Windows 11 is still vulnerable to the timing attack.

C. Large DNS Responses

The **third** eviction strategy forces the DNS resolver to drop multiple legitimate entries at once by resolving a domain that returns a large number of entries, again with a small TTL. In contrast to cache-hole-punching, this strategy uses large responses to evict legitimate entries from the cache, instead of eviction entries. This avoids filling the entire cache with individual DNS requests. However, even with DNS-over-TCP [23], the maximum size of a DNS response is limited to 65 535 B due to the 16-bit length header field. Even with a very short domain name 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 for domains with up to 16 characters. Thus, it is not possible to evict the entire cache in a single request using this primitive.

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 long-TTL entries. Afterward, 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. Afterward, 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 few individual 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. However, we found that the default-configured DNS server for our cloud servers 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.

D. Error-based Eviction

Our **fourth** primitive exploits the error-handling behavior of `systemd-resolved`. When the DNS response contains an error, e.g., by trying to return more entries than allowed by the intermediate DNS server, `resolved` will retry resolving the domain name. After three retries, `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 `SERVFAIL`. For these servers, the DNS cache is evicted after approximately 79.8 ms. Other servers do not respond to the query, causing `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, `resolved` will try to resolve the domain again. In our experiments, if the responses remain invalid, `resolved` will repeat this retry-and-switch process for multiple minutes, clearing the DNS cache each time. For DNS servers supporting extended DNS errors [52], `systemd-resolved` version 256 or later detects the error code and does not switch to the fallback DNS server. However, many systems, such as Ubuntu 24.04 LTS, still ship with systemd version 255 or lower, making them vulnerable to this primitive with any DNS server.

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.

Eviction Loop Recovery. To avoid the constant eviction caused by the retry-loop, the attacker can reply with a valid DNS response after the DNS cache has been flushed. This allows `systemd-resolved` to correctly resolve the domain after switching to the fallback DNS server, stopping the eviction loop. 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, using 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, making this a likely configuration.

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 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 the default browser DNS cache size is 1000 entries for Chromium and 800 entries for Firefox [8], [72], we insert 1000 attacker-controlled domains to occupy it fully with long-TTL entries, ensuring the browser

TABLE III
AVAILABLE EVICTION STRATEGIES FOR `systemd-resolved`.

Primitive	Availability RCE	JS	HTML	Eviction Time	Config-Dependent
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[2]	yes[3]

[1] DNS Server allowing arbitrary response size.
[2] Eviction time depends on upstream DNS behavior.
[3] Fallback DNS server configured, `systemd-resolved` version <256.

evicts target domains before attacker-controlled ones. As a result, subsequent genuine DNS requests propagate to the system-wide DNS resolver. Because the browser caches these target domains again after resolution, we repeat the eviction step before each measurement. Using the technique from Section V-C, the attacker can use CSS to fill the browser cache before performing the measurements, enabling cache eviction in a scriptless scenario. During a measurement with multiple target domains, the long TTLs of the attacker-controlled entries ensure that genuine entries evict each other, causing accesses to the system-wide DNS cache each time.

Concurrent Work. Moav et al. [67] exploit the network state partitioning feature in modern browsers to bypass the browser DNS cache. By measuring from different origins, they can force browser cache misses without having to evict the browser cache. This technique is effective and faster than our eviction primitive, but requires the attacker to control multiple domains and the ability to change origins during the attack. However, if those capabilities are available, this technique would increase the performance of our end-to-end attack as well.

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 is mitigated on newer versions of `systemd-resolved` when the upstream DNS server supports EDE [52].

VII. CROSS-VM END-TO-END ATTACK

In this section, we evaluate the end-to-end cross-VM attack, tracking the host's DNS cache from an unprivileged attacker in a VM, combining the native code execution measurement from Section V with error-based eviction from Section VI.

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` VM using `virt-manager`, representing the attacker, with a default configuration, running Debian 12. In the VM, we execute unprivileged native code to monitor the DNS cache of the host system.

The host uses a consumer-grade Internet connection (download: 299.9 Mbit/s, upload: 52.2 Mbit/s, latency: 10.7 ms), and the home router advertises itself as the DNS server with two IPv6 fallbacks, which is the default configuration provided by the ISP. 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 VM, 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 [2], `snee.la`, `hannesweissteiner.com`, and `asdf.com`. At the beginning of the victim's timeslice, the victim randomly chooses 8 domains, resolves them in a random order, and saves the results to a file with a timestamp. The victim then waits for the next timeslice for the next iteration.

Attacker. Inside the VM, the attacker executes an unprivileged Python script. At the beginning of the attacker's timeslice, it measures the execution time of `socket.gethostbyname()` for all 103 domains. We use 10 parallel threads to reduce the measurement's runtime. The attacker uses 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, and likely differ from host to host depending on system configuration and performance. Finally, the attacker evicts the host's DNS cache using error-based eviction to prepare for the next iteration.

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.

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%)

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 positives, 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 (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 $539.78\,\text{ms}$ 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 $5\,\text{s}$ to evict the DNS cache. Our ISP provides two fallback DNS servers. As `gethostbyname` is blocking, the call takes a total of $15\,\text{s}$ 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 $79.8\,\text{ms}$. Summarizing, the time required for an iteration, consisting of measurement and eviction, is $5.5\,\text{s}$ for our specific case, and $0.58\,\text{s}$ in a generic case. To optimize, the attacker could evict the cache asynchronously to increase the granularity of the measurements, as the remaining $10\,\text{s}$ 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 $20\,\text{s}$ with a blind spot of $5\,\text{s}$. 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 VM. The same attack can also be performed between two VMs, 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 VM.

VIII. JAVASCRIPT ACCESS DETECTION

In this section, we evaluate the end-to-end attack performance using JavaScript, combining the `fetch` JavaScript measurement presented in Section V with the error-based primitive and the browser DNS cache bypass from Section VI.

Setup. The setup consists of a host system representing the victim, which is connected via Ethernet cable to the same consumer-grade internet connection as described in Section VII. During the experiment, the network is not isolated, which means that other devices in the network can access the Internet normally, and induce noise on the network latency. 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 VI 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. 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 measurement 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. 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 VI. After $5\,\text{s}$, the measurement cycle is repeated, enabling the attacker to monitor the victim's activity continuously. The total duration of one cycle is approximately $25\,\text{s}$.

Evaluation. We evaluate the attack in two scenarios: with and without DNSSEC, testing the access-detection rates for ten common domains. 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 50 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. For our 10 tested domains, we achieved an average F_1 score of $82.86\,\%$ with DNSSEC enabled, compared to $78.89\,\%$

TABLE V
JAVASCRIPT END-TO-END ATTACK RESULTS

Domain	DNSSEC ✗	DNSSEC ✓
amazon.com	81.63 %	91.67 %
pornhub.com	85.71 %	80.77 %
reddit.com	86.49 %	97.78 %
wikipedia.com	95.24 %	91.67 %
Macro-average	78.89 %	82.86 %

We show a selection of individual domains, as well as the macro-average over all 10 domains. While DNSSEC generally increases the attack's accuracy, some domains show a decrease in performance, likely due to network noise in our non-isolated setup and measurement inaccuracies.

without it. For some domains, such as pornhub.com, the performance of DMT decreases slightly when DNSSEC is enabled. We attribute this to other network traffic in our non-isolated setup, which can affect network latency [29]. 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 81.63 %, 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 most evident for reddit.com, which showed the fastest average access times among the tested domains, loading in under 20 ms on average when the DNS resolution was cached. At the same time, reddit.com showed a large increase in accuracy when DNSSEC was enabled, reaching an F_1 score of 97.78 % (*i.e.*, *one single misclassification*), compared to 86.49 % without it. This is expected, as with fast server response times, the increase due to DNSSEC has a larger relative impact on the total request time. In contrast, wikipedia.com has significantly slower server response times, averaging approximately 120 ms for cached requests. However, the response times are very consistent, leading to high accuracies both with and without DNSSEC. The slight drop in F_1 score when DNSSEC is enabled can again be attributed to measurement inaccuracies due to network noise in our non-isolated setup.

Our findings show that DMT works in an end-to-end setting without native code execution and represents a significant threat to user privacy, especially when DNSSEC is active.

IX. 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 is always 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, VMs, 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, increases 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 V-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 even 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. Especially for VPNs, which often have features to avoid leaking DNS activity (to local networks), it is not obvious that they still leak information via the local DNS cache. 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), placing even more trust in the operators of these systems. Furthermore, DNS caches on these remote systems are more likely shared across multiple users, or even public, and thus can be used for different attacks.

X. 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 certain VM 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 or serving exploits tailored to a victim environment, e.g., their EDR solution or installed applications, before the web page has finished loading.

Ethics Considerations

Except for the experiments on private Internet connections in Section V-D and Appendix C, all of our long-running experiments were performed on virtual private servers in a data center, to not impact any end users. 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 network timing (e.g., 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 did not collect any user data. Thus, even if a user accessed our measurement website unintentionally, their DNS measurements cannot be linked to their identity.

Responsible Disclosure. We disclosed our findings to the systemd team on 2025-10-11, the Chromium team and Mozilla on 2025-10-15, and Apple on 2025-11-18.

The systemd team stated that systemd-resolved is a local-only service, and does not consider the security boundary between users on the same system as part of the threat model. Thus, they do not consider DMT a security issue. We followed up, stating that our browser-based attacks allow *any website* to query the local DNS cache, but received no further response.

The Chromium team acknowledged our timing attack, but stated that there is no practical mitigation that does not introduce drawbacks that are worse than the issue itself. Disabling the DNS cache entirely would introduce significant performance penalties, while forcing Chromium to use only the internal resolver by default would break setups that require the system resolver, such as enterprise environments using custom DNS configurations.

The Mozilla team also acknowledged our attack, agreeing that activity tracking using cache side channels is bad. They also stated that they do not see a way for Firefox to prevent the underlying issue without introducing significant performance penalties. However, they are thinking about implementing throttling mechanisms once the browser DNS cache is full, to make browser cache-eviction attacks harder. They see the responsibility with the system DNS (*i.e.*, , systemd) resolver to mitigate eviction techniques to alleviate the issue.

Apple responded that they are investigating the issue, but did not provide further details.

In conclusion, all four of our contacted vendors acknowledged our findings, but none of them plan to implement any mitigations at the current time.

Acknowledgements

We thank Sudheendra Raghav Neela for letting us query his server for experiments. This research is supported in part by the European Research Council (ERC project FSSec 101076409), the Austrian Science Fund (FWF project NeRAM 10.55776/I6054), and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85). Additional funding was provided by generous gifts from Google.

References

[1] Y. Afek, A. Bremler-Barr, S. Danino, and Y. Shavitt, "A Flushing Attack on the DNS cache," in *USENIX Security*, 2024.

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

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

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

[5] 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.

[6] 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.

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

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

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

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

[11] 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.

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

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

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

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

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

[17] Chromium, "Chrome Host Resolution," 2022. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/master/net/dns/README.md

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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

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

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

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

[31] Google, "Gaining security and privacy by partitioning the cache," 2020. [Online]. Available: https://developer.chrome.com/blog/http-cache-partitioning

[32] 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

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

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

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

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

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

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

[39] 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.

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

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

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

[43] 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.

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

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

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

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

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

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

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

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

[52] W. Kumari, E. Hunt, R. Arends, W. Hardaker, and D. Lawrence, "RFC 8914: Extended DNS Errors," 2020.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

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

[60] 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.

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

[62] 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.

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

[64] 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.

[65] 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.

[66] 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.

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

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

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

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

[71] Mozilla, "Firefox DNS over HTTPS," 2025. [Online]. Available: https://support.mozilla.org/en-US/kb/firefox-dns-over-https

[72] ——, "StaticPrefList.yaml," 2025. [Online]. Available: https://github.com/mozilla-firefox/firefox/blob/ba6106a65317ab93cae85888ec656a36219cf47a/modules/libpref/init/StaticPrefList.yaml#L14324

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

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

[75] 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.

[76] 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.

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

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

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

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

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

[82] 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.

[83] 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.

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

[85] 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.

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

[87] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," in *ESORICS*, 2019.
[88] 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.
[89] 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.
[90] 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.
[91] P. Sirinam, M. Imani, M. Juarez, and M. Wright, "Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning," in *CCS*, 2018.
[92] P. Sirinam, N. Mathews, M. Rahman, and M. Wright, "Triplet Fingerprinting: More Practical and Portable Website Fingerprinting with N-shot Learning," in *CCS*, 2019.
[93] 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.
[94] 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.
[95] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Robust smartphone app identification via encrypted network traffic analysis," *TIFS*, 2017.
[96] 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.
[97] 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.
[98] T. Wang and I. Goldberg, "Improved Website Fingerprinting on Tor," in *WPES*, 2013.
[99] D. Wessels, W. Carroll, and M. Thomas, "RFC 9520: Negative Caching of DNS Resolution Failures," 2023.
[100] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security*, 2014.

APPENDIX

A. Experiment Timing

Most of our experiments were performed before the publication of concurrent work [67]. However, performed additional experiments during the revision process, when the results of the concurrent paper were already public. The following table lists the experiments that were performed after the publication of [67]. Experiments not listed in the table were performed before the publication of [67].

TABLE VI
EXPERIMENTS PERFORMED AFTER THE PUBLICATION OF CONCURRENT WORK [67].

Experiment	Section
Firefox Verification	Section VI-E, Figure 12
300 Mbit/s JavaScript End-to-End Attack	Section VIII
300 Mbit/s JavaScript Latency Measurement	Section V-D, Table I
MacOS DNS Cache Timing	Appendix B
300 Mbit/s latency histograms	Appendix C
VSCode activity & Copilot detection	Appendix D

B. MacOS DNS Cache Timing

We performed an experiment, evaluating the feasibility of DMT on macOS. We measured the DNS cache timing on an Intel MacBook running macOS Sonoma (14.7.4), in a native code setting using Python. We empirically determined that the DNS cache size is approximately 2 000 entries. We found that we could evict legitimate entries by filling the cache with 2 000 or more random requests. This allows us to perform a similar evict-and-reload attack on macOS. Figure 9 shows the results of our experiment. We see a clear separation between hits and misses, indicating that DMT is also feasible on macOS. However, due to the closed-source nature of macOS, we did not achieve fully reliable eviction, causing some cache hits even though we evicted the cache before the measurement.

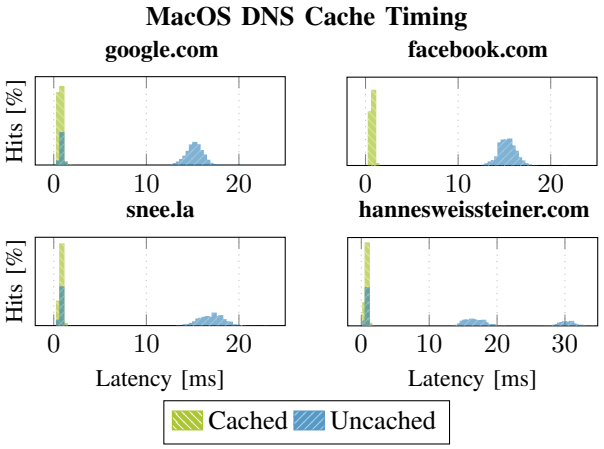

Fig. 9. Histogram of domain resolution latencies for cached and uncached domain names on macOS, using individual DNS requests to evict the DNS cache. The misses that still fall into the cached region are caused by unreliable eviction of the DNS cache. Still, there is a clear separation between hits and misses for all tested domains. Because we only evict using our eviction primitives, the histogram also proves the effectiveness of our eviction primitives on macOS. Interestingly, we did not measure any failed evictions for `facebook.com`, likely due to its lower TTL of 1 min compared to `google.com`'s 5 min, causing it to get evicted more reliably.

C. Latency Evaluation with 300 Mbit/s Connection

We performed additional latency measurements with a 300 Mbit/s cable Internet connection, to evaluate the impact of a typical higher-end home connection on our measurements. Since the measurements were performed on a shared cable connection, we experienced high latency variations throughout the day for browser-based measurements (cf. Figure 4). Thus, we do not show absolute latencies for browser-based measurements, since the attacker can account for these variations, but they look misleading in a histogram. Native measurements are less affected by these variations, as they do not require an HTTP request to trigger a DNS resolution.

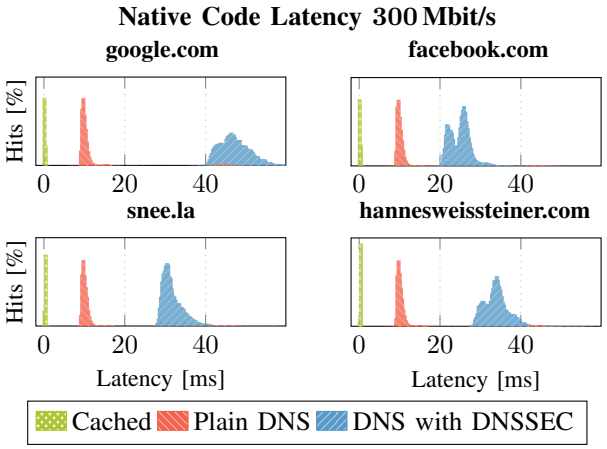

Fig. 10. Histogram of domain resolution latencies for cached and uncached domain names on a 300 Mbit/s Internet connection. As expected, we have a similar shape, but slightly larger separation compared to Figure 2.

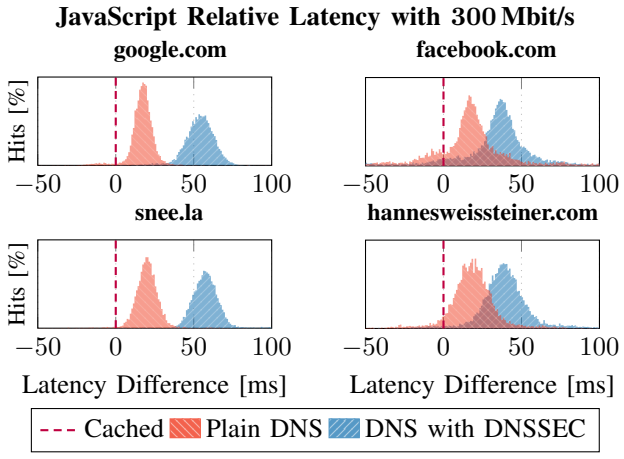

Fig. 11. Latency difference histogram for JavaScript measurements on a 300 Mbit/s connection. The increased network speed results in a clearer separation between cached and uncached DNS resolutions compared to Figure 8, indicating that the JavaScript primitive experiences less noise when using higher network speeds, and higher separation with higher latency connections.

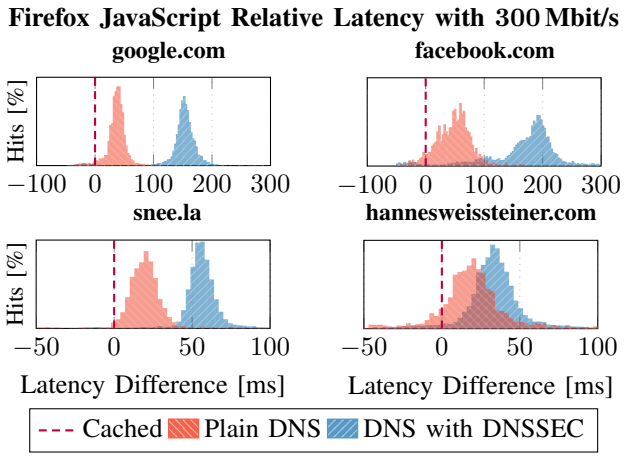

Fig. 12. Latency difference histogram for JavaScript measurements on a 300 Mbit/s connection, using Firefox. We experience significantly higher differences between cached and uncached websites compared to Chrome (Figure 11). We achieve a false negative rate of 5.89 % for facebook.com with DNSSEC enabled, and 7.75 % without DNSSEC.

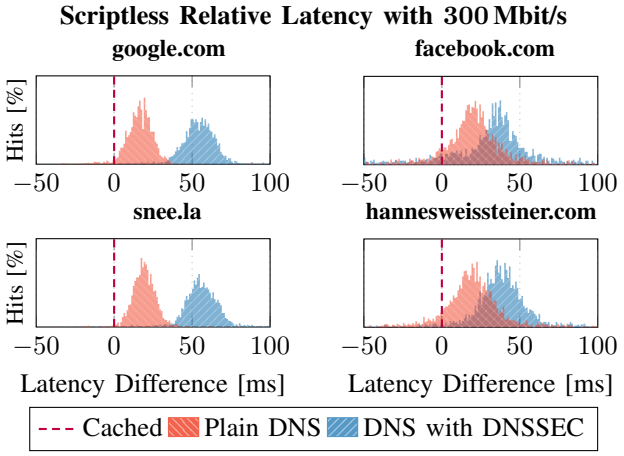

Fig. 13. Latency difference histogram for Scriptless measurements on a 300 Mbit/s connection. Similar to the JavaScript measurements in Figure 11, we observe a clearer separation between cached and uncached DNS resolutions compared to Figure 6, indicating the scriptless attack is more reliable in realistic home network conditions, compared to datacenter-grade connections.

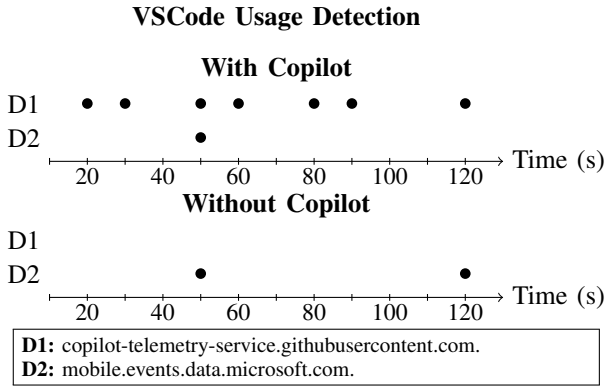

Fig. 14. DNS Cache hit trace of a user typing in Visual Studio Code with Github Copilot enabled and disabled, while measuring the DNS cache every 10 s. A dot indicates that a domain was cached within the last 10 s before the measurement. In both traces, the user takes a break from typing approximately 35 seconds after the start of the trace. In the trace without Copilot, the user stops typing after approximately 110 seconds. We observe that VSCode sends telemetry data approximately 10 seconds after users stop typing to **D2** (visible at second 50). When Copilot is enabled, autocomplete requests trigger frequent requests to **D1**, allowing for reliable detection of Copilot usage while the user is typing.

D. Activity Detection in VSCode

As an additional case study, we demonstrate how DMT can be used to detect user activity in Visual Studio Code. We identify two domains that are accessed while a user is interacting with VSCode. `mobile.events.data.microsoft.com` is used by VSCode to send telemetry data about user activity. We observe that VSCode sends telemetry data when the user stops typing for a few seconds. The second domain, `copilot-telemetry-service.githubusercontent.com`, is accessed frequently when GitHub Copilot is enabled, as Copilot continuously provides autocomplete suggestions while the user is typing. We perform two measurements where a user types in VSCode for approximately two minutes, once with Copilot enabled and once with Copilot disabled. The user takes a short typing break at approximately 35 seconds into the measurement in both cases, stopping shortly before the end of the measurement. During both measurements, we probe the DNS cache every 10 s to check whether the two domains are cached, evicting it afterward. The results are shown in Figure 14. When Copilot is enabled, we see frequent accesses to `copilot-telemetry-service.githubusercontent.com` while the user is typing, allowing an attacker to reliably detect Copilot usage. When Copilot is disabled, we can still detect whenever the user stops typing due to VSCode's telemetry requests. Thus, DMT can be used to infer detailed user activity beyond website visits.