UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

Doctoral Programme in
Information Engineering and Computer Science

Security Implications of Web Caching

Matteo Golinelli

Advisor
Prof. Bruno Crispo
Università di Trento

February 2026

Acknowledgements

When I started my PhD, 4 years seemed like a really long time. Now that I look back at them, though, I realize that they went by so quickly it's almost frightening. But when I count the number of people I've met over the years, it starts making sense again. It's now time to thank them: let's name some names.

First of all, I would like to thank Bruno for giving me this opportunity and for his guidance throughout this journey. Convincing him of the validity of my ideas was often the clearest indication that they were worth pursuing.

Special thanks to Ali for sharing his passion with me and always finding the time to discuss ideas and guide me, no matter how busy his life already was.

Thanks to Kaan for his guidance and for doing the best impression of a reviewer B early on, setting us on a good path right away.

Thank you to my thesis reviewers, professors Luigi Lo Iacono and Stefano Calzavara, for their careful reading of the thesis and for their constructive feedback.

Thanks, Luca, for being such an inspiring instructor and convincing me to start a PhD without even knowing what it really was.

Thanks to my colleagues, whom I can call my friends, for making my time here special and for ensuring that no day passed without having some fun together. I am convinced that better office mates are impossible to find. One of the greatest downsides of a PhD is how easily colleagues move on, sometimes far, far away.

Thank you, Helena. Spending my days with you is the best thing I could ask for.

Infine, grazie alla mia famiglia per avermi sempre fornito i mezzi e il supporto per raggiungere i miei obiettivi.

Abstract

The World Wide Web relies heavily on caching to improve performance and scalability, yet the security aspects of this mechanism remain poorly understood. This thesis investigates the security posture of web caches following three incremental steps: web cache detection, exploitation, and uncovering of novel attack primitives. First, we introduce methodologies to detect web caches using response headers, timing analysis, and subtle header variations, comparing their effectiveness and limitations. Building on this foundation, we present large-scale techniques for detecting vulnerabilities such as Web Cache Deception (WCD) and cache poisoning. We focus on understudied vulnerabilities for which no automated detection tools exist. Our empirical analysis includes the largest WCD study to date, identifying 1,188 vulnerable domains and challenging prior assumptions about its real-world severity. We then explore the broader security implications of cache misuse, showing how WCD can be chained with other web vulnerabilities to create complex attack vectors enabling data leakage and supply chain compromise, and how caching of security tokens can severely impact the security of web users. Finally, we introduce Web Cache Overflow (WCO), a new attack primitive that exploits imprecise cache keying to degrade cache performance and cause Denial of Service. Overall, this work provides a comprehensive exploration of web cache vulnerabilities, from foundational detection challenges to large-scale exploitation and mitigation, and serves as a basis for further research in this critical area. Through these contributions, we advance the state of the art in web cache security through systematic detection methodologies, large-scale vulnerability analysis, and the discovery of new attack vectors, accompanied by open-source tools to foster further research and defensive development. Our findings underscore the need for improved security practices in web caching and provide actionable insights for both researchers and practitioners.

Keywords

web caches, security vulnerabilities, web cache deception, cache poisoning, denial of service

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The World Wide Web has evolved from its first days, when everything on it was hypertextual files, to an extremely complex and heterogeneous system. The simple original client-server architecture has been enhanced with various types of "middleware", i.e., software and hardware components placed on the client-server path that serve different purposes. These can be used for security, such as firewalls and Intrusion Detection Systems, and for performance improvements, such as web caches and proxies.

1.1 The Role of Web Caches

Web caches and Content Delivery Networks (CDNs) store frequently accessed web objects to serve them quickly to the next visitors, enabling faster loading times and meeting the performance expectations of website visitors. However, the introduction of web caches in website architectures can have unexpected consequences on their security posture, potentially leading to the onset of security vulnerabilities and dangerous behaviours.

The fact that web caches, and more generally middleware, can be dangerous is a known fact since the very first days of their introduction, to the point that a group of experts at the time was hoping that they were a short-lived hack [60]. That is because such components interfere with core Internet principles, often breaking protocol semantics, violating the end-to-end design, and causing subtle and hard-to-debug failures in applications that rely on transparent network behaviour [61]. In fact, web caches can violate the semantics of HTTP by serving content in ways that diverge from the expectations set by the protocol. For instance, a cache may respond to a GET request with a previously stored response that is no longer valid, despite the origin server

indicating via headers (e.g., `Cache-Control: no-cache`) that the resource should be revalidated. In more subtle cases, caches may incorrectly reuse responses across users or sessions, leading to privacy leaks or inconsistent behaviour. These violations break the assumptions of correctness and freshness that applications and users rely on, and can be particularly damaging when caches are deployed transparently, outside the control of the origin server or client, but can also lead to severe security vulnerabilities, as we will see throughout this thesis. Moreover, one of the foundational architectural principles of the Internet is the end-to-end (E2E) principle, which argues that most features, such as reliability and security, should be implemented at the endpoints rather than in the network [106]. Middleware breaks this principle by embedding logic into the network itself, logic that attempts to inspect, modify, or optimize traffic. This not only adds complexity and statefulness to the network, but also creates dependencies on behaviour that was meant to be transparent [54]. At the same time, many argue that the E2E principle is too simplistic and limiting for the web, and that a hop-by-hop model would be preferable, allowing for efficient caching middleware to enable great bandwidth and, therefore, cost savings [54]. The hop-by-hop model embraces the presence of intermediaries — such as caches and proxies — which can improve latency, reduce server load, and enable policy enforcement closer to users. While this approach inevitably sacrifices some purity and transparency, it provides concrete benefits that make it more suitable for the web's operational realities [55]. In many cases, the web has evolved toward this model not out of design preference, but out of necessity.

Nowadays, (almost) no one seriously advocates for the elimination of caches and middleware; someone arguing that the web would be better off without them would most likely be dismissed as uninformed or out of touch with modern web architecture. At the same time, especially accounting for their omnipresence, thoroughly assessing and considering their security should be a top priority.

Web caches are one of the most important elements in the modern web ecosystem. Users expect low loading times and high performance from websites, and even short waiting times may result in a visitor leaving the page early or for e-commerce websites not to convert a sale. The idea that slow website loading times may cause users' frustration, profit loss, and general dissatisfaction is not new, and has been documented since the early years of general use of the Internet [112, 102, 113]. More recent studies show that even loading times of three seconds may drive visitors away from websites [114]. Nowadays, the loading time is used as a ranking factor by Google and other major search engines to sort websites in the results page, meaning that slower websites are less likely to be placed among the first positions in the search query response [145].

1.2 Attacks on Web Caches

Web caches front web servers and temporarily store and quickly serve frequently accessed objects. That translates to reduced load for servers and better performance for clients. Web caches can be placed at any step of the web client-server architecture. The security community is no stranger to attacks targeting web caches. These often fall under one of two categories: poisoning caches with an exploit payload to be delivered to unsuspecting clients, or tricking the cache into storing confidential information, which is then publicly exposed on the Internet. Attacks date back to the early 2000s, and the fundamental techniques have not significantly changed over the years, but the attack surface and damage potential *have*.

Content Delivery Networks, which are globally distributed Internet overlay networks made up of caching reverse proxies, have become a ubiquitous component of many online systems that have stringent scalability, availability, and performance requirements. Official deployment figures published by three major CDN vendors, Akamai, Cloudflare, and Fastly, give us a glimpse of the vast amount of traffic proxied via these web caches [4, 22, 39]. A recent measurement by Guo et al. shows that 74% of the Alexa Top 1k websites utilize a CDN for delivery [57]. As of November 2025, BuiltWith estimates that of the top 10k, 100k, and 1M websites they observe, 57.48%, 54.13%, 49.65% are behind a CDN, respectively [14]. Combined with many other stand-alone caching proxies (e.g., Squid, Varnish [131, 118]) and caching servers (e.g., Apache, NGINX [6, 91]) sprinkled along the Internet, it is evident that web caches are rapidly becoming critical infrastructure. That, in turn, considerably increases the likelihood and impact of a web cache attack.

1.3 Problem Statement

As previously established, web caches are a fundamental component of the modern web architecture, introduced to reduce latency, alleviate server load, and improve user experience. By storing and reusing previously fetched resources, caches play a critical role in meeting the scalability and performance demands of modern web services. From Content Delivery Networks to reverse proxies and browser caches, these systems are now deeply embedded in the infrastructure of nearly every high-traffic website.

Despite their critical importance, the security implications of web caches have long been overlooked. While other components of the web stack, such as browsers, application logic, and databases, have undergone extensive security scrutiny, caches have

received comparatively little attention. This gap in focus is problematic, especially considering that caches are not passive components: they actively modify the behaviour of HTTP communications by intercepting, storing, and serving content. This means that their presence can interfere with standard security assumptions, especially regarding authentication, authorization, and data integrity.

Cache-related vulnerabilities, such as *Cache Poisoning* and *Web Cache Deception*, demonstrate how adversaries can exploit subtle misalignments between caching logic and application behaviour. These attacks do not necessarily rely on flaws in the cache or the origin server taken alone, but rather on the complex interplay between the two. Attackers have learned to weaponize even the smallest differences in the interpretation of requests by different HTTP components, with severe consequences on the security of the web.

One of the fundamental obstacles to the study of web cache vulnerabilities is the lack of visibility into caching behaviour. Unlike server headers or application logic, which can often be examined directly, cache behaviour tends to be opaque. Identifying whether a resource is served from a cache or directly from the origin server is often non-trivial. This is further complicated by the lack of standardized headers for conveying cache status: while some implementations include diagnostic headers, such as `X-Cache`, their presence, semantics, and reliability vary significantly across deployments.

This opacity hinders both manual analysis and the development of automated tools. Without accurate detection of caching behaviour, it becomes difficult to determine whether a vulnerability exists, let alone scan for it at scale. Moreover, many caches are selectively deployed, serving only static files, or operating only under specific network conditions, making their detection a prerequisite for any meaningful security assessment. This is worsened by CDNs, which are run by third-party providers that generally don't share their internal configurations and functioning details, making it even harder to reason about their caching behaviour. Ethical considerations also limit the ability to perform intrusive testing on live systems, further complicating the study of cache vulnerabilities.

Even when such vulnerabilities are identified, their impact is frequently underestimated. Cache-based attacks can be chained with other common web issues, such as URL normalization flaws, inconsistent access control policies, misconfigured routing logic, and injection vulnerabilities to create novel attack paths. These composite attacks can lead to severe consequences, including data leakage, cache poisoning at scale, privilege escalation, and denial of service. As such, understanding the broader security implications of caching requires not only detecting individual flaws but also reasoning

about how they interact with the rest of the application and infrastructure.

This thesis investigates these problems and possible solutions through the following research questions:

Q1. How can we detect the presence of web caches and distinguish cached responses from non-cached ones?

Q2. Can we leverage cache-detection techniques to automatically identify web cache vulnerabilities at scale?

Q3. What is the real-world prevalence of such vulnerabilities on popular websites?

Q4. What is the broader security impact of web cache vulnerabilities when combined with other common web issues?

1.4 An Analytical Model of Web Cache Security

This thesis investigates multiple classes of web cache vulnerabilities. To keep these results comparable and to make clear how they relate to a single overarching construct, we rely on a simple analytical model of web caching. This is not a threat model in the traditional sense; rather, it abstracts away attacker capabilities and instead captures the mechanics by which caches store and reuse HTTP responses, and the specific points where those mechanics can diverge from web application semantics.

We view a web cache as a stateful component placed between clients and an origin server. For each request, the cache derives a cache key, looks up the key in its store, and either serves a stored response (cache `HIT`) or forwards the request to the origin (cache `MISS`). Upon receiving an origin response, the cache decides whether it may store it, and for how long, based on a cacheability policy. This decision is derived from HTTP semantics and possibly local cache rules. This abstraction is described in detail in Chapter 2.

Web applications implicitly partition responses by scope. Some objects, such as static assets, are global, while others depend on user state (e.g., cookies or authentication), content negotiation, geolocation, device class, or one-time security tokens embedded in HTML. We can define caching as safe when the cache key and the cacheability policy preserve these boundaries, i.e., when any two requests mapped to the same cache entry are guaranteed to be equivalent with respect to the application. Security problems arise when the cache stores or serves content across scope boundaries. For example, a

response whose scope is narrower than the cache key becomes shared (confidentiality leaks), or an attacker can influence which response is stored under a key that other clients will later use (integrity violations).

Under this model, the vulnerabilities studied in this thesis are instances of a single primitive: an attacker manipulates untrusted, cache-relevant input (such as the URL structure, query strings, headers, and sometimes body fields) to change the cache key or influence the cacheability policy such that the cache stores or serves a response outside its intended scope.

We can classify these vulnerabilities along three dimensions, each corresponding to a specific aspect of cache behaviour. First, **cacheability errors** occur when the cacheability policy is too permissive, allowing responses that should not be cached to become cacheable under crafted requests. This enables confidentiality leaks, such as Web Cache Deception vulnerabilities and the mistaken caching of security tokens. Second, **keying and parsing mismatches** arise when the cache key is imprecise or inconsistent across components. Attacker-controlled request elements that are unkeyed, or normalised differently, enable cache poisoning and cache-poisoned denial-of-service, while overly sensitive keys enable systematic cache busting. Third, **state and resource abuse** happens when an attacker leverages the cache's resource management policies to exhaust cache capacity or evict legitimate content. By exploiting key imprecision to create many distinct entries for semantically identical content, an attacker can force the cache to evict useful entries, leading to Denial of Service.

These dimensions are not mutually exclusive: many vulnerabilities involve multiple aspects of cache behaviour. However, this classification helps clarify the root causes and exploitation techniques, providing a structured framework for analysis.

1.5 Contributions

This thesis advances the state of the art in web security by addressing the overlooked but critical attack surface introduced by web caching systems. Our contributions span both theoretical understanding and practical tooling, with a strong emphasis on empirical validation. At a high level, we provide novel techniques for detecting caches, identifying vulnerabilities, and understanding their impact in the wild.

First, we introduce a suite of methodologies for detecting the presence of web caches and for distinguishing cached and non-cached responses. This is a fundamental prerequisite for studying cache behaviour and identifying vulnerabilities that depend on it. Our approaches are designed to cover a wide range of real-world scenarios. One method

leverages cache status headers, such as `X-Cache`, even in the face of their inconsistent and non-standardized use. A second method is designed for scenarios where cache status headers are entirely absent or unreliable, relying instead on timing patterns analysis. For completeness, we also explore alternative detection strategies that, while effective only in narrow conditions, help clarify the boundary of what is and is not observable in current cache deployments.

Building on this foundational capability, we develop a set of techniques and tools for the fully automated detection of specific web cache vulnerabilities. We focus on understudied vulnerabilities for which no automated detection methodology is available. These tools are designed to work without prior knowledge of the target application and are able to uncover a range of flaws, including Web Cache Deception (WCD) and Cache Poisoning Denial of Service (CP-DoS). Our goal is to bring cache vulnerability discovery closer to the level of automation and scalability that other classes of web vulnerabilities have achieved.

Among our most impactful contributions is a novel detection methodology for Web Cache Deception. Unlike prior work, which required specific preconditions, such as the need for authentication, our approach is general and applies to any website. In a comparative experiment on 404 domains, we show that our method is capable of identifying more than 100 vulnerable sites, while the previous state-of-the-art technique is limited to only 18. Thanks to its generality and robustness, our method enables the largest WCD study to date, conducted over the Alexa Top 10k. In this study, we discovered 1,188 vulnerable domains. We complement this quantitative analysis with case studies that challenge common assumptions about WCD: we show that attacks targeting non-authenticated pages can be highly damaging, and that the consequences of WCD go well beyond the typical leakage of personal data.

In addition to WCD, we study other classes of cache-based vulnerabilities through large-scale empirical analyses. These studies reveal that such vulnerabilities are not rare anomalies, but systemic issues present in a wide range of websites and caching infrastructures. Through our scanning efforts, we discover and responsibly disclose several previously undocumented vulnerabilities and misconfigurations, affecting high-profile and widely used web services.

We next investigate the security impact of cache vulnerabilities and explore how cache-based attacks can be chained together with other vulnerabilities. Contrary to the perception that such vulnerabilities are isolated edge cases, we show that they can be combined with other common flaws, such as improper access control or URL normalization issues, to form complex attack chains. These chains can significantly amplify

the impact of individual vulnerabilities, enabling attackers to bypass protections, leak sensitive data, and degrade service availability in unexpected ways.

Finally, we introduce a new attack primitive, which we term *Web Cache Overflow* (WCO), that builds on the cache busting technique previously explored in this thesis. This attack exploits imprecise cache keys to inject large volumes of redundant cache entries, effectively filling the cache and forcing eviction of legitimate content. WCO demonstrates how adversaries can use the cache infrastructure itself as a vector for Denial of Service, using only minimal resources.

In our work, we support the reproducibility and adoption of our research by releasing open-source tools for cache detection, vulnerability scanning, and measurement. These tools are intended to assist both researchers and practitioners in understanding and improving the security posture of web caching systems, both in laboratory and production environments.

Together, these contributions provide a comprehensive exploration of web cache vulnerabilities, from foundational detection challenges to large-scale exploitation and mitigation. We believe this work will serve as a basis for further research in this under-explored area and help push caching systems toward more secure and transparent designs.

1.6 Dissertation Outline

The remainder of this thesis is structured as follows.

In Chapter 2, we provide the necessary background on the web, web caches and their functioning, and known vulnerabilities related to them. We focus on Web Cache Deception (WCD), cache poisoning, and HTTP request smuggling, as they are the most relevant to our work.

Next, in Chapter 3, we present several techniques to detect the presence of web caches and distinguish cached from non-cached responses in black-box deployments. We discuss strategies based on response headers lookup, timing patterns analysis, and also explore alternative detection methods.

In Chapter 4, we focus on cacheability errors and keying and parsing mismatches. Building on the detection techniques presented earlier, we develop automated methods to identify web cache vulnerabilities at scale. We introduce a novel approach for detecting Web Cache Deception (WCD) vulnerabilities that does not require authentication, prior knowledge of the target application, and manual intervention from the tester. We also discuss techniques for identifying Cache Poisoning Denial of Service (CP-DoS)

vulnerabilities. We use these methods to conduct large-scale empirical studies on the prevalence of these vulnerabilities in popular websites on the Internet.

Chapter 5 investigates the security impact of web cache vulnerabilities. We explore how cache-based attacks can be chained with other web vulnerabilities to create complex attack vectors. We also analyse the causes and consequences of the mistaken caching of security tokens.

Finally, in Chapter 6, we introduce a new attack primitive, Web Cache Overflow (WCO), which exploits imprecise cache keys to fill the cache with redundant entries, leading to Denial of Service. We discuss the mechanics of the attack, its implications, and potential mitigation strategies.

We conclude the thesis in Chapter 7, summarizing our contributions and outlining directions for future research in the field of web cache security, providing a high-level overview of the key findings and their implications.

Chapter 2

Background and Related Works

In this chapter, we provide the necessary background to understand the concepts and technologies that are relevant to our work. We start with an overview of the web, its architecture, and how it operates. We then delve into web caches, their purpose, and how they function within the web ecosystem. Finally, we discuss vulnerabilities related to web caches, particularly focusing on Web Cache Deception (WCD) attacks.

2.1 The Web

The web is a collection of resources that can be accessed by users through the Internet. The web is a decentralized system, meaning that no single entity controls it, and resources are distributed across many servers. Resources on the web can be accessible to anyone, or only to specific authenticated users. To navigate the web, users employ web browsers, which interpret and display the content of web pages and other resources. The web uses a *client-server* architecture, in which a *client* requests resources in which they are interested in to a *server*, which responds with the requested resources. Web pages are formatted using the *HyperText Markup Language* (HTML) markup language, which defines their structure and layout.

2.1.1 HTTP and HTTPS

The exchange of resources between servers and clients happens using an application layer protocol called *HyperText Transfer Protocol* (HTTP), which in turn uses the *Transmission Control Protocol* (TCP) at the transport layer. Originally, HTTP was a *clear-text* Protocol, meaning that the data exchanged between clients and servers was not encrypted. However, with the advent of security concerns, a secure version

of HTTP called *HTTP Secure* (HTTPS) was introduced, which uses encryption to protect the data exchanged between clients and servers. HTTPS works by using the *Transport Layer Security* (TLS) protocol to encrypt the data exchanged between clients and servers, ensuring that the data remains confidential and secure during transmission. The first version of HTTP, known as HTTP/1.0, was introduced in 1996, later followed by the standardization of HTTP/1.1 in 1999, which introduced several improvements, such as persistent connections and chunked transfer encoding [44].

HTTP/2 Hypertext Transfer Protocol (HTTP) is the application-level protocol that is used to transfer data on the web. HTTP/2 was released in 2015 and was the first major update since HTTP/1.1, which was first published as a proposed standard in 1997. HTTP/2 is based on TLS over TCP and maintains the semantics of HTTP/1.1, but it changes the way that the data is transferred. While HTTP/1.1 was a plain-text human-readable protocol, HTTP/2 is a binary protocol. HTTP/2 implements optimized mappings of the HTTP/1.1 semantics to enable efficient use of the connection, allowing multiple concurrent requests and responses to be multiplexed over a single TCP connection and compressing the headers. Requests multiplexing consists of the organization of HTTP messages in streams, that are bidirectional sequences of frames with the same identifier, generally representing a request-response pair. Frames are the smallest protocol unit in HTTP/2: they can be of different types (e.g., data, headers, settings) and have an ID that identifies the stream to which they belong [125]. Web servers process the requests as soon as they have all the frames, and send the response as soon as it is generated.

HTTP/3 In 2022, the Internet Engineering Task Force (IETF) published HTTP/3, which differs from HTTP/2 in that it uses the QUIC transport protocol instead of TCP. QUIC is a transport protocol that runs on top of UDP and provides multiplexing, encryption and congestion control directly. HTTP/3 was developed to solve the problems caused by the fact that TCP has no visibility over HTTP/2 multiplexing. Therefore, some features of HTTP/2 are delegated to QUIC in HTTP/3 (i.e., multiplexing and flow control), while others are implemented on top of it [10]. According to Internet surveys, HTTP/3 surpassed HTTP/2 in terms of usage in March 2025 [134].

2.2 Web Caches

Even with troves of personal and sensitive data traversing the Internet, a disproportionately large slice of traffic is made up of content available for general consumption. Websites are rapidly growing in size due to an increasing number of objects necessary to render each page, such as images, videos, fonts, JavaScript files, and style sheets [115]. Furthermore, latency-sensitive media streaming services and bandwidth-hungry big file downloads have become commonplace. Repeated transfers of such objects can quickly get costly for both servers and clients, and even impact the overarching Internet infrastructure involved in traffic delivery. All of these can put an enormous strain on an origin server's network resources if left unaddressed. Web caches are designed to address this problem and have therefore become an essential element of the modern web.

Web caches are positioned between a client and an origin server to temporarily store frequently accessed objects. These effectively act as proxies between clients and origin servers, directly serving the responses that they previously cached. When a client requests a resource, the proxy intercepts the request and checks if it already holds a cached copy of the response. If it does and is not expired, the cache delivers it directly to the client. Otherwise, the cache forwards the request to the origin server. When the cache receives the response, it forwards it to the client and, if it matches some pre-configured criteria, caches it for future visitors. This prevents unnecessary repeated data transmissions, reducing round-trip time for the requester, load for the server, and the overall traffic volume for the Internet infrastructure.

Multiple caches can be present on the delivery path, starting with client-side caches (e.g., implemented inside a web browser), layers of reverse proxy servers acting as man-in-the-middle caches, and finally caches co-located with the origin server, forming cache hierarchies. Content Delivery Networks (CDNs), that provide performance and security services over globally distributed networks of reverse proxies (i.e., *edge servers*), similarly cache content as a core capability and have become pervasive [57, 14]. CDNs have the added benefit of placing content physically closer to end-users, significantly reducing the distance data needs to travel, and consequently improving website loading times, leading to a better user experience.

CDNs offer numerous options for website administrators to configure the caching behaviour according to their needs. For example, caching decisions can be made based on the request endpoint, file extension, query string parameters, presence of a cookie, request headers, response content type, or a complex combination of many similar parameters [33, 20, 34]. More recently, major CDNs have also started to offer *edge*

computation capabilities, enabling website operators to make these decisions programmatically [27, 3, 36].

In this thesis, we use the term *web cache*, or simply *cache*, to refer to any type of public server-side web cache. These caches are designed for storing static objects that do not have confidentiality requirements, whereas dynamically generated content that includes personal or sensitive information for each different client must be fetched from the origin afresh with each request. It is important to point out that one should not conflate *static* content with *public* content. For instance, public web pages may still contain unique, sensitive parameters dynamically generated for each visitor. We leave client-side caches out of scope, as attacks on such caches (i.e., a client attacking itself) do not constitute a valid threat model for the attacks and vulnerabilities that we investigate. We also do not further explore niche uses of private caches, but instead focus our discussion on the far more prevalent and essential public caches.

2.2.1 Cache Control

HTTP provides a standard mechanism for origin servers to signal to the caches on the path whether a response may be stored, and for how long. This is done via the `Cache-Control` response header. The directives that can be included with this header define the cacheability of an object, its expiration time, and the revalidation actions.

RFC 9111 states that caches *MUST* respect `Cache-Control` directives [42]. However, this is often not the case in practice, as also documented by prior academic work (e.g., [87]). Website structures and caching policies may frequently change, which makes constant maintenance of `Cache-Control` directives to reflect those changes an operational burden for website operators, especially in large enterprise infrastructures. Therefore, popular cache technologies offer configuration options to disregard `Cache-Control` headers, but instead implement *cache rules* centrally at the cache. These cache rules can typically be made as general or specific as needed, allowing website operators to craft them via domain specific languages or regular expression matches on virtually any part of a request or its corresponding response (e.g., [21, 26]).

2.2.2 Cache Keys

Once a response is determined to be cacheable, the cache derives a *cache key* for the respective stored object. The cache key acts as a unique index into the cache store, enabling the cache to easily check whether it already holds a copy of the object upon

receiving subsequent requests for the same resource. That is, the cache store implements an associative array as an abstract data type, keyed with the cache key.

Cache keys are n-tuples (or hashes of those n-tuples) that may consist of any element of an HTTP request, including the method, path, individual query string parameters, headers, and their values, or structured data fields in the body payload. Request elements that are included in the cache key are called *keyed*, and the rest *unkeyed*. A typical default cache key is a simple 2-tuple consisting of the full URL (e.g., [132]), but additional elements may also be keyed for more complex situations, for instance, the `Origin` header for CORS support (e.g., [25]). Website operators further customize the cache keys according to their needs.

A *cache purge* is the process of invalidating a cached object, meaning that the next request that matches that object's cache key will be a cache miss. That request will then be forwarded to the origin server, and the cache will store the new response under the same cache key. Web caches provide secure mechanisms for website operators to purge caches, typically through an authenticated web interface or an API. Arbitrary web clients should never be able to purge the cache, as this would facilitate Denial of Service (DoS) attacks, cache poisoning, and cache-timing-based side-channels.

2.2.3 Cache Status Headers

Web caches often add cache status headers to responses, indicating how they handled the corresponding request. This field may simply include a cache hit versus miss flag, or more detailed debugging information.

RFC 9211 is a proposed standard that aims to standardize how caches communicate this information, but sadly, it has not yet been widely adopted [93]. Instead, each web cache technology uses their own custom headers to this end, and these are not always officially documented. Mirheidari et al., however, compiled a list of the different header names and values observed when interacting with a selection of popular stand-alone cache servers and CDNs [88].

2.2.4 Cache Busting

Cache busting collectively refers to client-side techniques used for intentionally bypassing a cache and receiving a fresh copy of the requested object from the origin server. Cache busting is frequently used in web development, to rapidly test website changes without needing to purge any intermediary caches (which may not even be under the developers control) that may still hold stale copies of modified resources. Similarly,

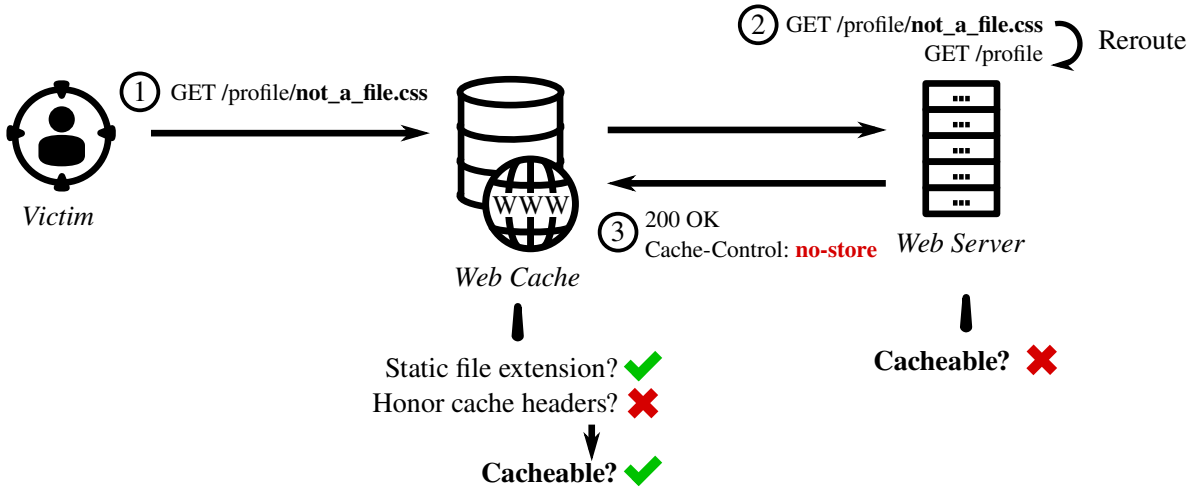

Figure 2.1: WCD in action. A social engineering victim clicks on a malicious URL, which in turn tricks a web cache into storing sensitive profile information, publicly exposing it on the Internet.

penetration testers often leverage cache busting to avoid poisoning cacheable endpoints with attack payloads.

According to RFC 9111, clients may attempt cache busting by including a "no-cache" directive in the `Cache-Control` header of their requests [42]. However, caches are not required to honour this request directive, and many production infrastructures in fact do not, due to the aforementioned security implications. Instead, practical cache busting techniques involve identifying keyed elements of a request that do not alter the response, and intentionally modifying those elements in requests to trigger a cache miss.

2.3 Web Cache Vulnerabilities

When caches are not properly configured, they can introduce vulnerabilities that can be exploited by attackers. Attacks targeting web caches date back to the early 2000s, and they mainly fall into two categories: poisoning caches with an exploit payload to be delivered to unsuspecting clients, or tricking the cache into storing confidential information which is then publicly exposed on the Internet. In this section, we provide an overview of two the most relevant web cache vulnerabilities: Web Cache Deception (WCD) and web cache poisoning.

2.3.1 Web Cache Deception (WCD)

Web Cache Deception (WCD) is an attack that exploits the request processing discrepancies between a web cache and an origin server, and subsequently tricks the cache into erroneously storing sensitive content. WCD was introduced by Omer Gil in 2017 [50, 51]. Below, we demonstrate the attack through a hypothetical case inspired by Gil's original proof-of-concept.

Figure 2.1 represents a typical deployment model where the origin application server is fronted by a cache. The cache server is configured to store frequently accessed static objects as determined by checking their file extensions. The attack begins when a miscreant crafts a malicious link containing the URL to a page with sensitive user profile details, but also appends to it an invalid path component that *appears* to be a static file. In this case, *"example.com/profile/"* is the legitimate page being targeted, and ***"not_a_file.css"*** is a reference to a non-existent style sheet. The attacker then distributes the resulting attack URL containing the WCD payload via social engineering channels, and the attack plays out as follows.

1. The victim clicks on the link and their browser issues the HTTP request for the resource. The web cache receives and promptly forwards the request to the origin server, being it a newly crafted URL.

2. The origin receives the request for the made-up resource and sees that the referenced style sheet does not exist. Therefore, it strips away the invalid path component, and reroutes the request to the *"/profile"* endpoint instead. This happens when the web server routes are configured to follow *clean URL* principles, which is a common practice in modern web frameworks [140], or when routes are defined using regular expressions. The origin server then generates the response containing the sensitive profile details, and sends it back to the cache. The server also indicates that the profile details should not be cached by setting the appropriate cache control headers in the response.

3. The web cache receives back the response and consults its caching rules. Oblivious to the request rewriting taking place at the origin, the cache finds a match indicating that *.css* extensions are cacheable. While there may be cache control headers present in the response, the cache is not configured to honour upstream headers. The web cache concludes that the response is safe to store. At this point, the sensitive content is publicly accessible under the URL *"example.com/profile/not_a_file.css"*.

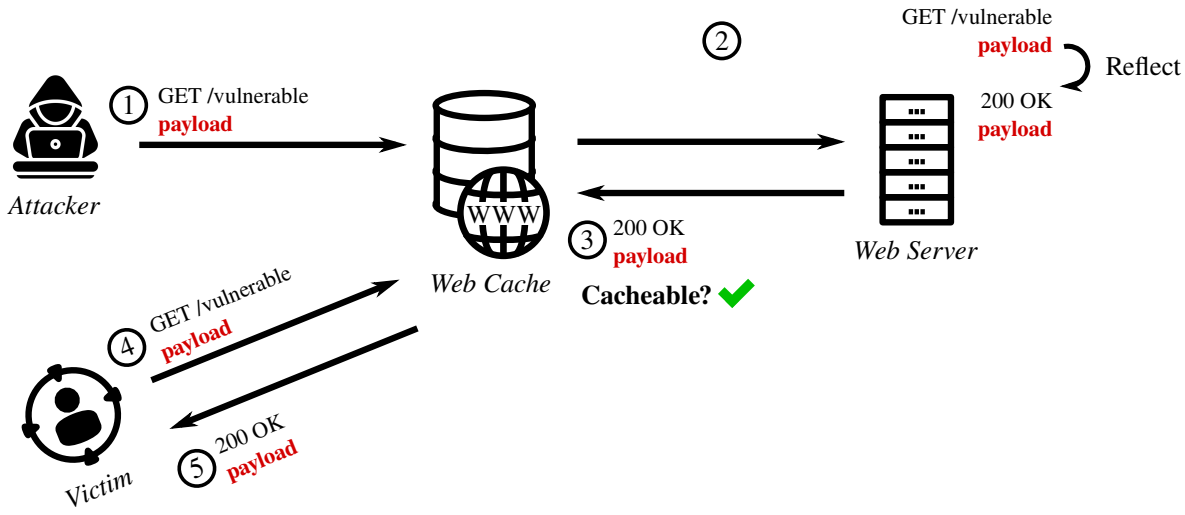

Figure 2.2: Web Cache Poisoning Attack Overview

This attack is possible due to the complex interactions between web caches, origins, and their administrators, which collectively lead to myriad potential HTTP processing discrepancies. For example, the request rerouting in Step 2 is a common behaviour implemented by web frameworks that follow *clean URL* principles, as opposed to treating URLs as filesystem paths [140]. However, this backend logic is invisible from the caching proxy's vantage point. Similarly, ignoring upstream cache control headers is common practice and sometimes the default web cache configuration [87]. For instance, in a large enterprise environment, where centralized management of caching rules is preferable to individually configuring web servers to return the correct headers. All in all, detecting and mitigating WCD is a non-trivial task, and neither application owners nor cache vendors are to individually blame; this is a complex system interaction problem.

2.3.2 Web Cache Poisoning

Web cache *poisoning* is a class of attacks that involves tricking a web cache into storing a malicious payload. This essentially escalates any reflected web application attack into a stored one, widely distributed to every client accessing the cache. Figure 2.2 illustrates a typical web cache poisoning attack. The attacker crafts a malicious request that contains an exploit payload, which is then reflected in the response that is sent to the cache. The cache stores the response, which includes the exploit payload, and serves it to all subsequent clients that request the same resource.

James Kettle presented a set of such attacks on popular caching proxies [73], and

more recently introduced more advanced attacks exploiting the cache key construction mechanisms used by these technologies [76]. In academic literature, Chen et al. exploited the inconsistent processing of the host header values in requests to the same effect [19]. Nguyen et al. proposed a different take on cache poisoning, employing erroneous negative caching (i.e., caching of error responses) as a means to block access to websites, resulting in a denial-of-service attack, called CPDoS (Cache-Poisoned Denial-of-Service) [92]. Liang et al. presented a systematic study of web cache poisoning in the wild, proposed a detection mechanism to identify vulnerable websites, and discovered 172 vulnerable websites on the Tranco top 1k [83]. Jia et al. instead focused on browser caches, finding that in 2015 virtually all major browsers were vulnerable to cache poisoning attacks [66].

2.3.3 HTTP Request Smuggling

An attack closely related to WCD and cache poisoning is *HTTP request smuggling (HRS)*. While HRS is not investigated in this thesis, it is worth mentioning it here as it is a well-known vulnerability that can also be exploited to poison caches and CDNs.

Web caches and proxies generally use the same connection with an origin server to send multiple HTTP requests, increasing efficiency and performance in general. The origin server parses the requests based on the headers to distinguish the boundaries of different HTTP requests. HTTP request smuggling (HRS) vulnerabilities arise when the proxy and the origin server disagree on the request boundaries, allowing attackers to send requests that are interpreted differently by the two entities. Consequences of this attack include JavaScript injection, cache poisoning and session hijacking. Since web architectures frequently comprise more than one entity in the path between the client and the origin server (for instance, when multiple reverse proxies sit in front of a back-end server with different purposes), the likelihood of such vulnerabilities arising is increased [100]. These dangerous interactions between proxies and origin servers are mainly caused by the fact that HTTP allows two different ways to communicate the messages' length: the `Transfer-Encoding` header and the `Content-Length` header. RFC 7230 states that the two should not co-exist in the same request and that when both are present, servers should prioritize `Transfer-Encoding` [45]. Some servers, however, do not support this header, and others can be tricked into not processing it using special obfuscation techniques [100].

The first documentation on HRS is a white-paper by Watchfire [84], but the vulnerability remained largely ignored for years due to the difficulty of exploitation and

the few consequences studied. Later, Kettle extensively studied this type of vulnerability, proposing new exploitation techniques and demonstrating the severity of their consequences [71, 75, 74]. Jabiyev et al. carried out a systematic study on 10 caches, CDNs, and server technologies combined to identify pairs that present discrepancies in the parsing of HTTP requests and are potentially affected by HRS vulnerabilities [65]. In [64], Jabiyev et al. present a fuzzer for HTTP/2 that they use to detect anomalies in HTTP/2-to-HTTP/1.1 protocol conversion performed by proxies and CDNs that result in severe security vulnerabilities. Jabiyev et al. present a guided differential fuzzer to test servers for HRS vulnerabilities, identifying critical vulnerabilities in several highly used server technologies [63]. Kettle in [77] and Lerner in [81] show how to exploit the downgrading of HTTP/2 requests to HTTP/1.1 performed by several proxy server technologies, which only communicate using HTTP/2 to the clients, while using the HTTP/1.1 in the origin-server connection. Müller et al. in [89] show how to use HRS to evade censorship.

Chapter 3

How to Detect Web Caches

This chapter is based on works previously published as:

Mirheidari, S. A., Golinelli, M., Onarlioglu, K., Kirda, E., & Crispo, B. (2022). Web Cache Deception Escalates!. In 31st USENIX Security Symposium (USENIX Security 22) (pp. 179-196).

Golinelli, M., & Crispo, B. (2024, September). Hidden Web Caches Discovery. In Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses (pp. 65-76).

Web caches play a critical role in modern web infrastructure, improving performance, reducing bandwidth usage, and enhancing scalability. However, their presence is often opaque to clients, and detecting whether a response has been served from a cache, or whether a cache exists at all, can be surprisingly difficult in certain cases. This opacity has significant implications for security research, since detecting caches is crucial to being able to assess their security. In this section, we outline the challenges involved in detecting web caches and explain why being able to do so is an essential capability for both defenders and analysts.

We start by discussing a previously available cache detection technique, highlighting its limitations and shortcomings.

We then present a novel methodology to detect caching based on *cache status headers*. These are specific response headers employed by web caches to communicate whether a response is coming from the origin server (a cache `MISS`), or if it was cached (a cache `HIT`). These headers are not standardized, therefore, different cache technologies might use different and custom header names and values. We present a technique to detect caching using lookups of cache status headers that we call *Cache Headers*

Heuristics (CHH). To evaluate its effectiveness, we crawl the top 10k most popular websites in the Tranco list [99]. We check whether they present the cache status of responses in the headers, finding that 79.1% of them do.

Techniques that detect cached responses based on cache status headers are not effective when these headers are missing, wrong, or use custom names and values that are not covered by the heuristics used. For this reason, we then present a novel methodology that uses timing analysis to distinguish between cached and non-cached responses, that can work against any web server, regardless of whether it communicates the cache status of responses using headers or not. Our methodology is based on repeatedly sending paired requests to a web server. The idea is that when the two requests reach a web cache at the same time and are processed concurrently, if only one of the two is cached, it will consistently return to the client first and faster. If, instead, both responses are coming from the origin server, they will arrive at the client with an inconsistent order and timing.

To control which responses will be served by the origin server and which by the web cache, we make heavy use of cache-busting techniques. Before developing our methodology, we carried out a preliminary experiment of different cache-busting techniques to identify the most effective against a higher number of websites in the wild. Combining all the techniques that we used, we are able to cache-bust requests on 84.3% of websites. Our methodology relies on the multiplexing functionality to pair requests together and send them in a single packet. We focus on HTTP/2 since it is currently the most adopted version, but the same methodology applies to HTTP/3 too.

We implement this methodology in a tool and perform a preliminary experiment to measure its accuracy compared with the CHH technique, estimating an accuracy of 89.6%. However, we observed that in the vast majority of the cases where this methodology classified the request as cached, while the cache status headers reported a cache MISS, it was due to an unpredictable behaviour of certain web cache technologies. These caches always report cache MISS when two requests are paired together, even if one of the two responses was a cache HIT. We randomly selected 100 websites and manually verified that in 82 of them, the wrong classification of this methodology was due to the behaviour mentioned above. We can therefore estimate that the real accuracy of our tool is higher, and the reported accuracy should be considered as a lower bound.

We use this methodology to estimate the prevalence of *hidden web caches* in the Tranco Top 50k, i.e., caches that do not advertise the status of their responses in the cache status headers, finding that 1.627 websites (5.8% of the 28.243 tested websites that supported HTTP/2) present a hidden cache.

Chapter Outline This chapter is organized as follows. We start discussing why automatically detecting web caches and distinguishing between cached and non-cached responses is important in Section 3.1. Section 3.2 presents our novel detection methodology based on lookups of response cache status headers, and Section 3.3 introduces a methodology that employs timing analysis to enable cache detection in scenarios where cache status headers are absent. Section 3.4 discusses other ways that could potentially be used for cache detection that present severe limitations that make them impractical for large-scale analyses. Finally, in Section 3.5, we conclude with a high-level comparison of different techniques.

3.1 Why is it Necessary?

From a security perspective, detecting the presence of caches is a necessary step for identifying and analysing cache-related vulnerabilities. Many classes of attacks, such as cache poisoning and cache deception, depend fundamentally on how intermediate caches behave. If a cache is present but undetected, it becomes impossible to evaluate whether a web application is vulnerable to such threats. This limitation hinders the development of automated vulnerability detection tools, which must first establish an accurate model of the underlying infrastructure before conducting security analyses.

Despite its importance, detecting web caches is a non-trivial problem due to a lack of standardization and consistency in how cache-related information is exposed. Most cache systems communicate the cache status of objects using custom response headers such as `X-Cache`. These headers vary widely in naming conventions, semantics, and even presence across different caching technologies.

To address this fragmentation, RFC 9211 has been proposed as a standard for caches to communicate how a request was handled [93]. It defines a more uniform mechanism for indicating whether a response was served from cache or by the origin server. While this represents a step toward interoperability, it has not been widely adopted. Consequently, relying on RFC 9211 alone is insufficient for general-purpose detection.

The situation is further complicated by the existence of what we term *hidden caches*, intermediaries that perform caching but provide no explicit signal to clients. These caches may make no use of diagnostic headers, operate in proprietary appliances, or reside within Internet Service Providers' or corporate infrastructures. In such cases, conventional detection techniques based on response headers fail entirely, requiring more sophisticated approaches.

The lack of standardized mechanisms to communicate cache status, the poor adoption of proposed standards, and the existence of silent caches make cache detection a challenging technical problem. Overcoming these obstacles is a prerequisite for any comprehensive effort to secure and analyse modern web architectures.

In this chapter, we present novel ways to automatically detect web caches and infer the cache status of responses.

3.2 Cache Headers Heuristics

In this section, we present a novel methodology to detect web caches based on the presence of cache status headers in HTTP responses. We start by discussing related works, their strengths, and their limitations.

3.2.1 Background and Related Works

3.2.2 Methodology

Algorithm 1 Pseudocode of the Cache Headers Heuristics technique.

```
1: Input: response
2: for all header in response.headers do
3:     header ← lowercase(header)
4:     if 'cache' ∈ header or 'server-timing' ∈ header then
5:         if 'hit' ∈ header or 'cached' ∈ header then
6:             return 'Cache HIT'
7:         else if 'miss' ∈ header or 'caching' ∈ header or 'stale' ∈ header then
8:             return 'Cache MISS'
9:         end if
10:    end if
11: end for
```

The *Cache Headers Heuristics* (CHH) inspect HTTP response headers to heuristically determine whether a request is served from the origin server or a web cache. Web caches often transform responses by including a header that indicates to the client the result of the cache lookup. However, as previously discussed, this mechanism is not standardized, and cache technologies implement their proprietary headers (e.g.,[38, 23, 101]). We performed an exploratory crawl of the Internet before this work, supplemented that with vendor documentation, and compiled a list of header fields and values returned by popular web caches. We present these results in Table 3.1.

Table 3.1: Cache lookup status headers as specified in RFC 9211 and as used by popular CDNs and web caches. Note that some CDNs and caches use multiple headers to indicate cache status.

CDN / Cache	**Header Name(s)**	***Hit* value(s)**	***Miss* value(s)**
RFC 9211	`Cache-Status`	`hit`	`fwd=miss`, `fwd=*-miss`, `fwd=stale`

Akamai	`server-timing`
	`X-Cache`
	`X-Cache-Remote`
		`desc=HIT`
		`TCP_HIT`
			`desc=MISS`
			`TCP_MISS`

CDN77	`X-Cache`	`HIT`	`MISS`
Cloudflare	`cf-cache-status`	`HIT`	`MISS`
CloudFront	`x-cache`	`Hit from cloudfront`	`Miss from cloudfront`
Fastly	`X-Cache`	`HIT`	`MISS`
Google Cloud	`cdn_cache_status`	`hit`	`miss`
KeyCDN	`X-Cache`	`HIT`	`MISS`
Azure	`X-cache`	`TCP_HIT`, `TCP_REMOTE_HIT`	`TCP_MISS`
Apache, ATS	`X-Cache`	`HIT`	`MISS`
NGINX	`X-Proxy-Cache`	`HIT`	`MISS`
Rack Cache	`X-Rack-Cache`	`hit`	`miss`
Squid	`X-Cache`	`HIT from *`	`MISS from *`
Varnish	`X-Cache`	`HIT`	`MISS`
Unknown	`x-cache-info`	`cached`	`caching`

Note that the headers and their values show strong similarities between different caches. Namely, all headers we identified contain the term `cache`, and most values are either `hit` or `miss`. Therefore, instead of doing strict equality checks, we normalize the received headers and then perform keyword searches in them. Additionally, we perform some further checks for key-value pairs of headers that are not covered by the previous check; specifically `server-timing` as a header name for Akamai, and the `cached` and `caching` values observed in the responses of a set of websites whose cache technology could not be identified. In our exploratory study, we determined that this method works as well as enforcing strict checks, with two added advantages. First, this approach makes our detection more robust against minor format or structure differences

Table 3.2: Pervasiveness of cache status headers captured by the Cache Headers Heuristics technique in the Tranco top 10k.

Have Cache Headers	No Cache Headers	Total Sites
4300 (79.1%)	1137 (20.9%)	5437

in headers often observed in the wild, for example, due to man-in-the-middle devices that incorrectly transform requests, or version differences between caches. Second, it opens up the possibility for this technique to work correctly with sparsely used or private cache technologies that may be observed in large-scale experiments, provided that they follow the same conventions with their headers. Algorithm 1 provides a simplified pseudocode for the CHH technique.

3.2.3 Effectiveness

We now discuss the effectiveness of the CHH technique by measuring how pervasive the cache status headers captured by our algorithm are. To do this, we crawl the websites included in the Tranco top 10k domain list generated on April 7, 2025. [1] We limit our crawler to visit at most 10 pages on a maximum of 10 unique subdomains per website to avoid generating unreasonable traffic volumes. Table 3.2 presents the results of our analysis. Of the 10k domains that we visited, we were able to crawl only 5437 of them, while the rest did not respond to HTTP requests. Using the CHH, we detect cache status headers on 79.1% of the websites. Note that this means that we detected the presence of cache headers on at least one URL on a specific target website. The remaining 20.9% either does not use a cache or has no or unconventional cache status headers.

3.2.4 Limitations

While the Cache Header Heuristics technique proves effective in identifying web caches through the presence of cache status headers, it suffers some key limitations that impact its reliability and coverage in specific scenarios.

First, a significant limitation arises when cache status headers are entirely absent. This may occur not because a cache is not in use, but because it has been deliberately configured to be stealthy. Some web administrators may suppress such headers as a security measure, relying on obscurity to prevent attackers from detecting and targeting

[1] Available at https://tranco-list.eu/list/LJL44

intermediate caches or proxies. While this is certainly a bad security strategy, in these cases, our method is inherently unable to detect the cache, despite its presence and activity.

Second, the reliability of CHH depends entirely on the correctness and consistency of the headers. However, cache implementations are not always robust: misconfigured caches or software bugs may cause incorrect or misleading values to be inserted in the headers. Additionally, in adversarial scenarios, a cache may intentionally provide false information, either by omitting the headers selectively or by inserting misleading values to prevent accurate detection.

Third, as previously discussed, the lack of standardization across the ecosystem poses a considerable challenge. There is no universal naming convention for cache status headers or for the values they contain. As a result, some caches may use non-standard or proprietary headers that fall outside the detection patterns recognized by CHH. In fact, while we do observe a vast majority of cache technologies following similar naming conventions, custom implementations may use entirely different names or embed the cache status in unrelated headers, making them harder to recognize automatically.

In summary, while CHH is effective for detecting many common cache setups, it cannot offer full coverage. Its effectiveness is constrained by deliberate header suppression, incorrect or deceptive values, and lack of standardization—factors that limit its use as a universal or foolproof method for web caches detection. In the next section, we present a novel methodology that is able to overcome these limitations by using timing analysis.

Availability A Python implementation of the Cache Headers Heuristics is available on the authors' repository. [2]

3.2.5 Section Summary

The Cache Headers Heuristics technique is a novel method for detecting web caches based on the presence of cache status headers in HTTP responses. By analyzing the headers returned by web caches, we can infer whether a response was served from the origin server or from a cache. Our analysis of the Tranco Top 10k revealed that 79.1% of websites include cache status headers, making this technique effective for a large portion of the web. However, it has limitations, particularly in cases where cache

[2] https://github.com/golim/wcde

status headers are absent, incorrect, or non-standard. To address these limitations, we introduce a complementary methodology based on timing analysis in the next section.

3.3 Timing Analysis

We now present a novel methodology to detect caching using timing analysis. First, we supplement the background provided in Chapter 2 with a discussion of theoretical foundations of timing attacks and provide an overview of the related works that inspired our methodology.

3.3.1 Background and Related Works

To develop our timing analysis methodology, we take inspiration from timing attacks.

Timing attacks focus on indirect leaks of information; specifically, the time it takes a system to perform certain tasks. By measuring the time variations between the execution of different actions, attackers can potentially extract sensitive data from a system. Timing attacks are more effective when performed locally, due to the absence of network jitter and delay, but previous studies during the last twenty years have shown that they are a viable attack vector over the network too. Timing attacks leverage the unintentional side effects of a system's operation. This makes them particularly hard to detect and requires careful design and implementation of security measures in software systems to prevent time-based leaks.

Timeless Timing Attacks (*TTA*), first introduced by Van Goethem et al. [52], are a novel type of timing attacks that improve the accuracy and greatly lower the required number of requests. They are based on measuring the relative timing difference between requests that are processed concurrently by the web server, while classical timing attacks consist of independent measurements over the network. To make the two executions concurrent, this attack technique sends the two requests in a single packet, enabling attackers to observe all the timing differences greater than the network jitter introduced once the requests arrive at the server. These differences can include the delay introduced by the network card, decryption, and ordering of packets. Moreover, timeless timing attacks observe the response packets sequence number (which is monotonically increasing in TCP) to identify the request that the server finished processing first. To send two requests in a single packet, an attacker can exploit HTTP/2 and HTTP/3's multiplexing functionalities. This way, two paired requests will reach the server at the same time and, ideally, be processed concurrently.

Timing attacks have been known and used since 1996 when Kocher introduced them and showed how they could be used to find Diffie-Hellman exponents, factor RSA keys, and break other cryptosystems [78]. Initially, timing attacks have only been used to break cryptosystems locally [32, 108, 109]. Later, several studies demonstrated the practicality of timing attacks over the network, enabling extracting private keys from network servers [13, 1, 9, 12, 29].

Felten and Schneider show how to exploit a browser's cache from a malicious web page to determine if a user had recently visited another unrelated web page by issuing requests and checking if the time required to get the response is less than a threshold [41]. Bortz et al. show two types of timing attacks against websites: direct timing, where private information is leaked directly by the attacker measuring the response time from the server, and cross-site timing, where a malicious website obtains information on a different website from the user's perspective [11]. Jia et al. show how to infer the geographical location of victims using timing attacks, exploiting the browser's cache [67]. Gelernter and Herzberg bypass the same-origin policy and extract sensitive information by measuring the time it takes for the browser to receive the responses to search queries [49]. Van Goethem et al. show that modern browsers expose new side channels that can be used to acquire accurate timing measurements regardless of network contentions and analyse new browser features that can be exploited to obtain substantially more timings [128]. All major browsers implement defence mechanisms to protect against timing attacks based on lowering the resolution of the timing information; however, Schwarz et al. prove this approach ineffective and present new mechanisms to obtain absolute and relative timings [110]. Smith et al. propose attacks to leak the browsing history of victims [116], and Sanchez-Rola et al. show a way to fingerprint hardware devices timing the execution of cryptographic browser API functions that can be mounted remotely with a malicious website and JavaScript code [107]. Vanderlinden et al. use the `server-timing` header value, which generally provides server-side timing information accurate to the millisecond, to reduce the impact of jitter on remote timing attacks. They show that this enables significantly reducing the required number of requests for a successful attack [130]. Similarly, Vanderlinden et al. use timing information exposed in HTTP response headers by backend servers to reduce the jitter included in an attacker's sample. Specifically, they use the timestamp of responses included by web servers in the `Date` header to synchronize the attacker to the target server, improving classical timing attacks by reducing the number of requests necessary for a successful attack [129].

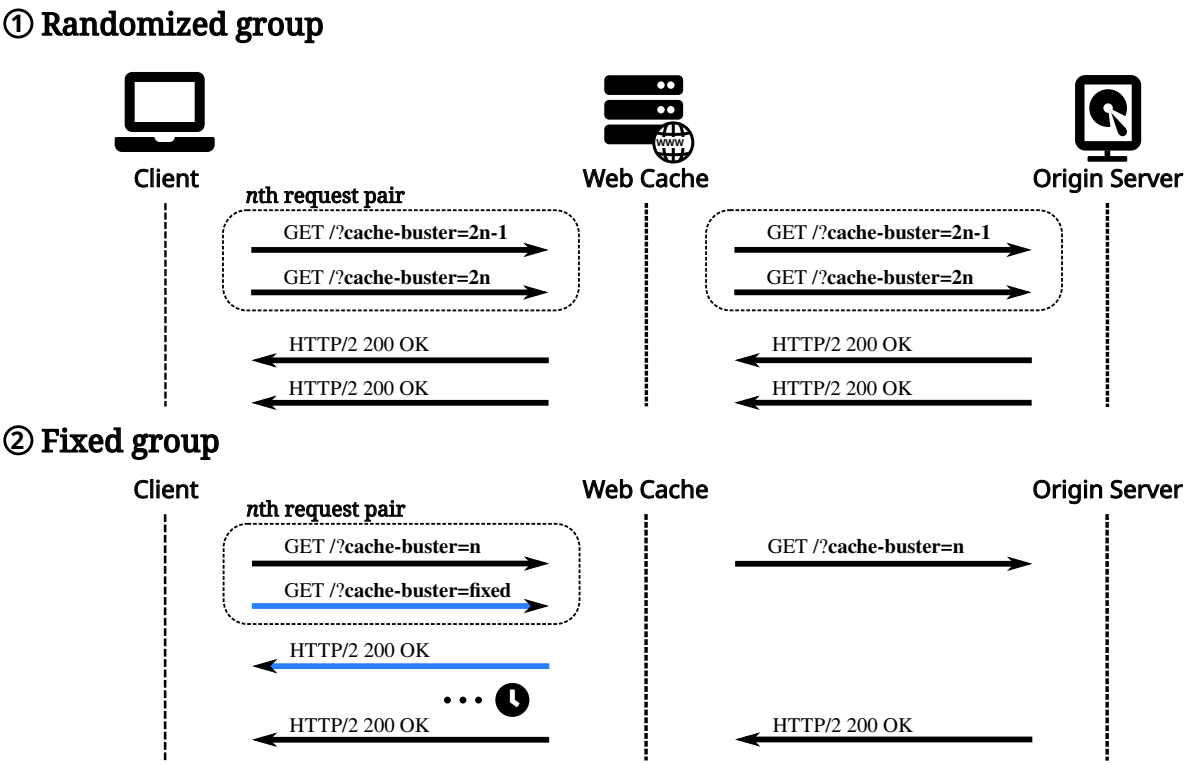

Figure 3.1: Overview of our cache detection methodology. Note that, for the Fixed group, we perform a request with fixed cache-busters before collecting the time measurements, so that the response should already be stored in the cache. We see that, in the Randomized group, all requests are forwarded to the origin server, and their order of arrival back at the client is inconsistent. For the Fixed group, instead, the response to the request with a fixed cache-buster is directly issued by the web cache, and will therefore consistently arrive at the client first and faster.

3.3.2 Methodology

This methodology does not rely on cache-communicated cache status headers and applies to all web servers that use newer versions of HTTP. It is composed of two main phases: a first phase where we collect the timing measurements, exploiting HTTP multiplexing and using cache busting, and a second phase where we analyse the timings collected in the first phase to infer if there is a cache or not.

Collection of Timing Measurements

In this first phase, we collect the measurements that will be used in the second phase to infer the presence of a cache. This phase is based on HTTP multiplexing and cache busting, respectively used to send the two requests within a single packet and ensure that the responses are served by the origin server and not by the cache. An overview

of the timing analysis that we perform is presented in Figure 3.1. The idea behind our methodology is to send:

1. `n` pairs of requests in a single packet where both requests have random cache busters (i.e., the responses should always be served by the origin server). We call this group of request pairs Randomized.

2. `n` pairs of requests in a single packet where the first request has a random cache buster and the second request has an already used cache buster (i.e., that was included in a previously sent request). In this way, the first response should be served by the origin server, and the second by the cache. We call this group of request pairs Fixed.

In our methodology, **n** is a constant number decided before the experiment. We observe the order of the responses and measure their arrival time difference. By comparing the order of arrival of responses and the time measurements of the two groups, we can determine whether there is a cache in the path from the client to the origin server. Note that we do not measure the absolute time required for responses to arrive, but only the relative time elapsed between receiving one response and the other, drastically decreasing the effect of network jitter on our measurements.

Cache Busting Since we do not have visibility over what fields are included in the cache key by the web caches, we introduce modifications in all the fields that we can modify without obtaining a different resource (e.g., we do not modify the path of the request and the `Host` header). Specifically, we introduce modifications in the *query string* by including new parameters with random names and values, and in the following request headers:

- `Origin`: all web caches should include the value of the Origin header in their response so as not to risk introducing Denial of Service vulnerabilities, as shown in [56]. Since an Origin is defined as the protocol, the host and the port, we do not modify these to avoid receiving a different response. Instead, we include a randomly generated path in the value of the Origin header, that will not influence the response but is likely included in the cache key.

- `User-Agent`: including the User-Agent in the cache key should not be necessary if the web pages correctly implement and use responsive designs. Since User-Agent values are extremely varied, including them in the cache key might lower the effectiveness of a web cache.

- `X-Forwarded-Host` and `X-Forwarded-Scheme`: used by proxies to communicate the original host and scheme to the origin server (that might differ). These headers are generally included in the default cache key of caches and CDNs.
- `X-Method-Override`: used by web frameworks to override the HTTP methods of requests. It is typically included in the cache key to avoid cache poisoning vulnerabilities.

Moreover, we include random modifications in the headers' values in the `Vary` response header. The `Vary` response header is used to communicate what parts of a request may induce differences in the server's responses (not including the method and the URI). Therefore, this header is typically used to communicate to a cache what values of a request must coincide with the ones of the cached response for it to serve the response (i.e., what values should be included in the cache key) [43]. For example, if a response contains the `Vary: Accept-Encoding` header, the cache should only serve the response if the `Accept-Encoding` header of the request is the same as the one of the cached response.

To evaluate the effectiveness of the different cache-busting techniques, we conducted an experiment on the Tranco Top 10k to identify what elements of HTTP requests are included in their cache key. For each website, we:

1. Crawl the website and identify a cached response using *Cache Header Heuristics*.
2. Send one request for each cache-busting technique tested; i.e., we send an HTTP request introducing a modification to a single element of the request.
3. Check if the response for each request is cached or not. If it is not cached, the modified element is included in the cache key and the cache-busting technique works, otherwise, the element is not part of the cache key.

Table 3.3 presents the effectiveness of different cache-busting techniques, based on our experiment on the Tranco Top 10k. We identified a cached response on 3494 websites. Collectively, with the techniques that we employed, we were able to cache-bust requests on 2946 websites. On the 3128 websites that included the `Vary` header in their responses, introducing modifications in the headers included in it effectively cache-busted the requests only on 616 websites. Therefore, 2512 websites include some header names in the `Vary` response header but do not configure their web cache(s) to include them in the cache key. In our experiments, we use all techniques combined to maximize the likelihood of effectively cache-busting the request.

Table 3.3: Results of our experiment on different cache-busting techniques on the Tranco Top 10k. Percentages are calculated over the total number of 3494 websites. Note that only 3128 websites included the `Vary` header in their responses. Collectively, with the techniques that we employed, we were able to cache-bust requests on 2946 websites.

Cache-busting technique	Cache busted
Query string	2112 (60.4%)
`Origin` header	817 (23.4%)
`User-Agent` header	78 (2.2%)
`X-Forwarded-Host` header	327 (9.4%)
`X-Forwarded-Scheme` header	329 (9.4%)
`X-Method-Override` header	338 (9.7%)
Headers in `Vary` header	616 (17.6%)
All techniques combined	2946 (84.3%)

HTTP Multiplexing To send two requests in the same TCP packet, we use the same methodology exploited by Van Goethem et al. to perform *timeless timing attacks* in [53], sending two `HEADERS` frames containing the two HTTP/2 requests in a single packet. Sending the two requests within a single TCP packet eliminates the effect of network latency and jitter, enabling timing measurements that are not influenced by them. Our methodology can also be implemented using HTTP/3 multiplexing, which enables simultaneously sending multiple requests over a single connection.

Reading the Timings

In the second phase, we read the time measurements collected in the first phase to infer whether the responses to the requests are coming from a web cache or if they originated from the backend server. To do this, we employ a statistical test. We use a `t-test` to determine whether there is a statistically significant difference between the means of the measurements of the Randomized and Fixed group of paired requests. If the difference is significant, we conclude that there is a cache in the path from the client to the origin server (we set the threshold of the p-value to 0.01). Otherwise, we conclude that there is no cache. We use the t-test as a classifier where, depending on a set threshold, we can classify requests as cached or not. Before performing the statistical test, to enhance its accuracy, we employ two heuristics to pre-process the data. First, we remove the time measurements with outlier values [3] to prevent delayed packets (that

[3]In particular, we compute the data's average and standard deviation and the absolute difference between the time difference and the average. If the absolute difference is lower than the standard deviation multiplied by a factor of 2, the data point is considered an outlier and removed. The factor

might be due to network congestion, server overload and other unpredictable conditions) from negatively influencing our classification. Then, we multiply the negative values in the Fixed group measurements by a factor of 5 when the group's mean is negative. We expect negative values when a response is cached because the request with a fixed cache buster is placed as second, and it will arrive before the response to the first request. We do this to amplify negative values, making them more significant and easing the statistical test classification. The factor value is based on our preliminary experiments and observations. In the Fixed group, when a cache is present, we expect negative timing measurements with low standard deviation.

Table 3.4 presents an example of timing measurements in a scenario where the second response of the Fixed group is coming from a cache (on the right) and one where all the responses are coming from the origin server. We can observe that, when one of the two responses of two paired requests is cached, the time measurements are all negative and close to each other with a low standard deviation.

was selected manually to remove the highest number of outliers while not losing too many data points.

Table 3.4: On top, a sample of the time measurements of a timing attack against a website that presents a web cache; below, a sample where the responses are not cached. We can observe that, where the responses in the Fixed group are cache, while the timings for the Randomized group are extremely variable, the timings for the Fixed group are consistent (i.e., negative with a low standard deviation). When instead all the responses are coming from the origin server, we see that both the Randomized and Fixed group's time measurements are inconsistent. Negative timings mean the response to the second request in the pair arrived first. *CHH* refers to *cache header heuristics* algorithm from [88]. For brevity, this example only shows 5 time measurements for each group of paired requests, while in all our experiments we sent 10. The two samples are from different websites.

Time measurements with cached responses

Group	Time diff. (ms)	Cache Status 1	Cache Status 2
Randomized	-60.09	MISS	MISS
	62.42	MISS	MISS
	-58.35	MISS	MISS
	67.32	MISS	MISS
	-77.45	MISS	MISS
Fixed	-600.95	MISS	HIT
	-504.63	MISS	HIT
	-591.15	MISS	HIT
	-516.49	MISS	HIT
	-536.35	MISS	HIT

Method	Cache Status
CHH	*Cache*
Statistical test	*Cache*

Time measurements with no cached responses

Group	Time diff. (ms)	Cache Status 1	Cache Status 2
Randomized	34.37	MISS	MISS
	97.29	MISS	MISS
	-486.03	MISS	MISS
	132.2	MISS	MISS
	-325.18	MISS	MISS
Fixed	-169.52	MISS	MISS
	12.2	MISS	MISS
	-409.99	MISS	MISS
	-31.29	MISS	MISS
	217.21	MISS	MISS

Method	Cache Status
CHH	*No Cache*
Statistical test	*No Cache*

3.3.3 Effectiveness

To test the effectiveness of our timing analysis methodology, we perform a preliminary experiment on websites in the Tranco top 10k list generated on January 29, 2024 [99] [4] that report the cache status of their responses in the headers to validate our hypothesis that timing analysis can be used to detect web caches and measure the accuracy of this technique. Our crawler is configured to run without authentication, meaning that it can only visit and test the web pages that are publicly accessible. Our crawler is developed in Python and uses the `requests` library to perform the HTTP requests. To avoid our requests being blocked by the websites, we use a real browser user agent, and we limit the number of performed requests per second. We only tested websites that support HTTP/2 in our experiments because of the availability of off-the-shelf libraries and because, at the time of this research, it was more widely adopted compared to HTTP/3; we leave the task of implementing an HTTP/3 version of our tool for future work.

This experiment aims at understanding if it is possible to infer the presence of a web cache by timing responses. Moreover, we measure the accuracy of the t-test in classifying whether a cache is present or not. To do this, this experiment only targets websites that present cache status headers in their responses and uses it as a ground truth for the cache status of the responses. If during the crawling phase we do not identify a request that gets cached, we issue a request to a non-existent path to obtain a *404 Not Found* response (which are generally cached). We do this because, in this experiment, we are not interested in the content of the cached response, but only in the response being cached.

During this experiment, we observed several websites issuing wrong or untruthful cache status headers that only reported cache misses, which lowered the accuracy value of the t-test. We investigated multiple cases where, based on the time measurements, a cache is almost certainly present to understand why the cache headers do not report the cache hits. We observed that this is mainly due to CDNs only reporting cache misses in the responses to paired requests, even when the responses are cached. Since CDNs are black boxes for us, we cannot provide possible explanations for this behaviour. To better estimate the real accuracy of the t-test in classifying cached responses based on time measurements, we then selected 100 websites for which the cache headers report only misses, while the t-test reports the presence of a cache, and manually verified if a cache is indeed present by sending requests in single packets and inspecting their cache

[4] Available at https://tranco-list.eu/list/QGN64

Table 3.5: The results of the preliminary experiment over the Tranco Top 10k. *The percentages of "Reachable", "Tested", "Analysed" and "Discarded" are calculated over the 10k websites from the Tranco list. The percentages of "Correct classification" and "Wrong classification" are calculated over the number of correctly analysed sites (1,946). 2,621 domain names could not be reached because they timed out, did not listen for HTTP requests or had other errors.

	Number of sites	Percentage*
Reachable	7,379	73.8%
Tested	2,289	22.9%
Discarded	343	17.6%
Analysed	1,946	19.5%
Correct classification	1,743	89.6%
Wrong classification	203	10.4%

status headers.

In this experiment, we crawl the websites starting from their homepage and, for each URL, we check if it reports the cache status headers. If it does, we collect the time measurements. To reduce the overhead caused by our tests on the targeted websites, we limit our crawler to test at most 10 URLs on at most 10 FQDNs for every root domain in the Tranco list. We stop our tests once we find one request that gets cached. Recall that we send two batches of paired requests, one where both paired requests are served by the origin, and the second where one of the two paired requests is instead served by the cache. Based on our initial experiments and observations, we set the number of request pairs **n** to 10 for each batch. Experimentally, we observed that a smaller number of request pairs deteriorated the accuracy of our tool, while increasing it did not provide significant improvements.

In the first batch, we expect both responses of paired requests to be cache MISS, i.e., they come from the origin, and in the second we expect one cache MISS and one cache HIT, i.e., one response from the origin and one from the cache. We discard from our data all the measurements with more than one wrong cache status in any of the two batches since we observed that a single error in the cache statuses does not compromise the overall quality of the classification. This number is a good compromise between the stress imposed on the tested websites and the accuracy of our classification.

Results Table 3.5 presents the results of our preliminary experiment. Of the 10k domain names in the Tranco Top 10k list, only 7.379 were reachable; the remaining 2.621 domain names timed out, did not respond to HTTP requests or redirected to

another domain included in the list. Of the reachable websites, we successfully test 2.289 websites which present cache status headers and support HTTP/2. We then discard 343 websites from our data due to responses with the wrong cache status (i.e., a cached response where we expect a response coming from the origin, or vice versa), resulting in 1.946 correctly analysed websites. On these, our methodology correctly identified the cache status of the responses on 1.743 websites (89.6%) and failed on 203 websites. However, as we mentioned previously, looking at the time measurements we hypothesize that a high percentage of these wrongly labelled measurements are due to the caches advertising a wrong or untruthful cache status, rather than our methodology failing to correctly identify it. For this reason, we select 100 websites and manually analyse them to validate our methodology's conclusion and find that 82 wrongly labelled measurements were due to wrong cache statuses being advertised by a web cache, and were therefore false negatives. Based on this data, we estimate that the accuracy of our technique in identifying the cache status of responses only based on timing analysis is higher compared to the one reported, which should be considered as a lower bound. This experiment confirms that timing analysis is highly effective in detecting cached responses and distinguishing them from responses originating from the backend server.

3.3.4 Hidden Web Caches

Now that we have a technique to detect caches without relying on cache status headers, we perform a large-scale measurement on the Tranco Top 50k list generated on January 29, 2024 [99] [5] to measure the prevalence of hidden web caches. For the websites that advertise the cache status of responses in the headers, we also check if the classification matches the reported status to better estimate the accuracy of our detection technique.

We again set the number of request pairs **n** to 10 for each group and limit our crawler to test at most 10 URLs on at most 10 FQDNs for every root domain in the Tranco list. For each URL, we collect the timing measurements and use the statistical test to infer the cache status of the response. We also check if the classification matches the cache status reported in the headers if they are present. We note that, due to the possible absence of cache status headers, we have no way of checking if the cache-busting techniques that we apply to the requests are effective or not. For this reason, it is possible that in some cases it will appear as though there is no cache, while in reality a cache is present and cannot be detected since no response is coming from the origin

[5] Available at https://tranco-list.eu/list/QGN64

server (i.e., there is no timing difference to observe between the Fixed and Randomized groups). In this experiment, we use the t-test as we did in the preliminary experiment and we apply the same pre-processing techniques to the time measurements.

Results Table 3.6 presents the results of our large-scale measurement experiment on the Tranco Top 50k. Of these 50k domain names, 10,841 could not be reached due to timeouts, redirects, and the absence of an HTTP server listening. We successfully visited 39,159 domains, 28,243 of which supported HTTP/2 and could therefore be tested with our methodology.

Of the 28,243 correctly tested websites, 10,543 advertise cache status headers in their responses, which we compared against the classification of our methodology. Using the cache status headers as a ground truth, 7.280 were correctly classified by our methodology (69.1% over the 10,543 websites that present cache status headers), while for the remaining 3,263 websites (30.9%), our methodology gave an incorrect classification. We hypothesize that the accuracy in this experiment is lower due to the lower popularity of the websites tested, which might use less efficient cache technologies and make less use of CDNs. For example, less popular websites might employ web caches placed on the same machine as the origin server. This results in smaller differences in the time measurements for cached responses and responses coming from the origin server. In this experiment, due to the larger sample size tested, we did not perform a manual validation of the results.

17,700 websites that we correctly tested did not advertise the cache status of their responses in the response headers (or used custom names and values for their headers that are not covered by the *cache header heuristics*). Of these, our methodology classified 1,627 websites as having a hidden web cache, and the remaining 16,073 as not having a web cache. Therefore, we estimate the prevalence of hidden caches at **5.8%**.

Table 3.6: The results of the large-scale experiment over the Tranco Top 50k. *The percentages of "Reachable" and "Tested" calculated over the 50k websites from the Tranco list. All the other percentages are calculated over the number of correctly analysed sites (28.243). 10.841 domain names could not be reached because they timed out, did not listen for HTTP requests or had other errors.

	Number of sites	Percentage*
Reachable	39,159	78.3%
Tested	28,243	56.5%
Present cache status headers	10,543	37.3%
Correct classification	7,280	25.8%
Wrong classification	3,263	11.6%
No cache status headers	17,700	62.7%
Cache	1,627	5.8%
No cache	16,073	56.9%

3.3.5 Limitations

The timing analysis technique we presented has several limitations that should be considered when using it for cache detection. First, it requires a higher number of requests compared to the Cache Headers Heuristics technique. In particular, in our experiments we need to send 20 requests (10 pairs) to each target object, while CHH only requires a single request. This means that the timing analysis technique is more intrusive and can generate more traffic on the target website. Second, the timing analysis technique has a false positive rate, which is not easy to estimate. This means that it can classify a response as cached when it is not, or vice versa. In our experiments, we observed that the false positive rate is lower than 10% in most cases, but it can vary depending on the target website and the cache technology used. Finally, the timing analysis technique relies on the assumption that the cache is faster than the origin server. This means that if the origin server reside on the same machine, or in the same location, as the cache, the time measurements might not be able to distinguish between cached and non-cached responses.

3.3.6 Ethical Considerations

During our experiments, we minimized the impact of our tests on the load of the targeted servers by limiting the number of requests performed as much as possible and by slowing our requests to no more than two paired requests every half a second.

Specifically, during the crawling phase, we limit our tool to visit at most 10 pages on at most 10 FQDNs, for a maximum of 100 requested web pages. In reality, as most of our tests were performed on a single page for each domain in the Tranco list, we stop the crawling phase once we identify a candidate page where we can successfully perform our tests. In both experiments, our timing analysis comprised two runs of 10 paired requests each, resulting in 40 requests sent. We argue that this number is a good compromise between the accuracy of the timing analysis and the excess load introduced on the servers by our tests.

3.3.7 Conclusion

In this section, we presented a novel methodology for detecting cached responses using timing analysis. This methodology overcomes the limitations of previous approaches that rely on cache status headers, which are not standardized and can be missing or unreliable. Our method applies to any web server that supports HTTP/2 or HTTP/3, regardless of its cache disclosure practices. We developed a timing analysis-based methodology that achieves an estimated accuracy of 89.6% in differentiating between cached and non-cached responses. We identified an uncommon behaviour where certain web caches only report cache MISSes for paired requests, even if one response is a cache HIT, highlighting the limitations of solely relying on cache status headers to detect caching. Using our methodology, we estimated that 5.8% of websites within the Tranco Top 50k that support HTTP/2 employ hidden caches that do not advertise their presence through cache status headers. Our findings demonstrate the effectiveness of our novel timing analysis methodology for cache detection. This methodology provides a valuable tool for security researchers and website operators to assess caching behaviours and identify potential security risks.

Availability A Python implementation of this cache detection technique is available as an open-source tool on the authors' repository. [6]

3.3.8 Section Summary

In this section, we presented a novel methodology for detecting web caches based on timing analysis. This technique does not rely on cache status headers and can be applied to any web server that supports HTTP/2 or HTTP/3. We showed that this technique is effective in differentiating between cached and non-cached responses, achieving an

[6] https://github.com/golim/hidden-web-caches-discovery

estimated accuracy of 89.6%. We also found that some web caches only report cache MISSes for paired requests, even if one response is a cache HIT, highlighting the limitations of relying solely on cache status headers for cache detection. Finally, we estimated that 5.8% of websites within the Tranco Top 50k that support HTTP/2 employ hidden caches that do not advertise their presence through cache status headers.

Table 3.7: Cache technologies and their identifying header names and values.

Cache	**Header Names**	**Header Values**

Akamai	`x-akamai-*`	`akamai`
		`akamaitechnologies`
		`akamaiedge`
		`AkamaiGHost`

CDN77	`x-cdn77` `x-77`	`cdn77`

Cloudflare	`cf-cache-status`
	`cf-ray`
	`cf-request-id`
		`cloudflare`

CloudFront	`x-amz-cf-pop`
	`x-amz-cf-id`
	`x-amz-*`
		`cloudfront`
		`cloudfront.net`

Fastly	—	`fastly`
Google	`x-google-*` `x-goog-*`	`1.1 google`
KeyCDN	—	`keycdn`
Azure	`x-msedge-*`	`azure`
Apache ATS	—	`apache` `ATS/`
Nginx	`x-nginx`	`nginx`
Rack Cache	`x-rack-cache`	`rack-cache`
Squid	—	`squid`
Varnish	`x-varnish`	`varnish`

3.4 `Date` Header

The `Date` header is a standard HTTP header that is used to communicate the date and time at which the response was generated by the origin web server. According to RFC 7231, "The "Date" header field represents the date and time at which the message was originated" [46]. In presence of a web cache, depending on the interpretation of the term "originated", it could mean the time when the origin server first served the object, or the time at which it was served by the cache. The use of the term here is ambiguous, and, to the best of our knowledge, there are no other documents specifying how this should be interpreted. This matter was discussed in a Hacker News thread [47], and we tend to agree with the original poster that the `Date` header should not be changed by web caches and proxies based on the wording of the RFC. That is because we argue that the cache is not originating the message, it is just sending a copy of an object that was originated in the past by the origin server–hence the name.

Consequently, if the `Date` header should not be changed by web caches and proxies, it should be possible to use it to detect whether a response is cached or not: if the `Date` header of a response is not changing between responses, the response is cached. Otherwise, the response is not cached.

To detect whether an object is cached or not, this technique works as follows. We request the object the first time and record the value of the `Date` header. We request the object a second time and compare the value of the `Date` header with the value previously recorded. If the value is the same, we can conclude that the request is cached. If not, the request is coming from the origin server.

This works in theory; however, depending on the interpretation that different caching technologies and CDNs give to the RFC, this might not be possible in practice.

Caches Documentation To test what technologies change the `Date` header, we first analysed the documentation of the most popular web caches. The list of caches that we investigated is presented in Table 3.8. We could find mentions of this only for Fastly and Apache Traffic Server (ATS):

- *Fastly.* According to the following documentation, Fastly changes the value of the date header: "If a `Date` header is present on a response when served by Fastly, we will update the value to the current time. If the header is not present, it is not added" [37].

- *ATS.* According to their documentation, ATS, and the Apache server in general,

do not change the value of the `Date` header: "where *date* is the date in the object's server response header" [31].

Empirical Analysis For the rest of the cache technologies and CDNs, we could not rely on their documentation to understand their behaviour. Therefore, we developed a simple crawler that detects which caches change the value of the `Date` and which do not. In this experiment, we use the Tranco top 1k generated on September 2, 2023. [7] The crawler starts from the homepage of the website and recursively visits the URLs found inside the web pages. We limit the crawler to visit at most 10 pages on 10 unique FQDNs. For each visited URL, the crawler uses the Cache Headers Heuristics to check if the response is cached. We continue until we identify a cached object, or we hit the limit. Once we identify a cached object, we need to determine the cache technology that is fronting the target website. Note that, for each cache technology, we validate our results by checking multiple websites that are fronted by the same cache technology to ensure consistency in the behaviour observed.

Identifying a Cache Technology To identify the cache that is fronting a specific website, we develop a heuristic algorithm that inspects the header names and values of HTTP responses, searching for specific identifiable information. Table 3.7 presents identifiable header names and values for the different cache technologies that we investigated. We also denylist three header names that frequently lead to false positives: `content-security-policy`, `content-security-policy-report-only`, and `access-control-allow-origin`. These headers are used by websites to enforce security policies and are not indicative of a specific cache technology.

Does the Cache Change the `Date` Header's Value? Finally, testing for the cache's behaviour proceeds as follows:

- We first issue two requests to the same URL and compare the value of the `Date` header in the two responses.
- If the values in the two responses are different and the CHH confirms that both requests are cached, we can conclude that this specific cache does change the date header.

[7] Available at https://tranco-list.eu/list/6JNPX

Table 3.8: Cache technologies and their behaviour regarding the `Date` header.

Cache Technology	Changes the Date Header?
Akamai	Yes
CDN77	Yes
Cloudflare	Yes
CloudFront	**No**
Fastly	Yes
Google Cloud	Yes
KeyCDN	Yes
Azure	Yes
Apache, ATS	**No**
NGINX	Yes
Rack Cache	Yes
Squid	**No**
Varnish	**No**

- Otherwise, we issue a cache-busted request and compare the `Date` header's value of the response with the value of the previous two. If the value is now different, we conclude that the cache investigated does not change the value of the `Date` header.

Table 3.8 presents the results of our investigation. Four of the 13 caches that we investigated do not change the value of the `Date` header: CloudFront, Apache and ATS, Squid, and Varnish. For these caches, using the `Date` as an indication of a cached object is a viable option. For all the other caches, this cannot be done, since the value changes with every response managed by the web cache, cached or not.

Availability A Python implementation of the `Date` header detection technique is available on the authors' repository. [8]

3.4.1 Section Summary

In this section, we investigated the use of the `Date` header as a potential technique to detect cached responses. We first analysed the documentation of the most popular web caches and found that only Fastly and Apache Traffic Server (ATS) explicitly mention their behaviour regarding the `Date` header. Fastly changes the value of the `Date` header, while ATS does not. For the other cache technologies and CDNs, we developed a crawler

[8] https://github.com/Golim/date-header

to empirically test their behaviour. We identified the cache technology fronting each website by inspecting the header names and values of HTTP responses. Finally, we found that four of the 13 caches we investigated do not change the value of the `Date` header: CloudFront, Apache and ATS, Squid, and Varnish. For these caches, using the `Date` header as an indication of a cached object is a viable option, while for all other caches, this technique cannot be used.

3.5 Chapter Summary

In this chapter, we presented three methodologies to detect web caches and distinguish between cached responses from non-cached ones. The first methodology, Cache Header Heuristics, relies on the presence of cache status headers in the responses to HTTP requests. We investigated the effectiveness of this technique by crawling the Tranco top 10k domain list and found that it can detect cache status headers on 79.1% of the websites. However, we discussed how its functionality is limited by missing, incorrect, or deceptive cache status headers. The second methodology uses timing measurements to infer the presence of a cache without relying on cache status headers. This technique is more complex and requires sending multiple requests, but it can detect caches that do not communicate their status in the headers. We call these caches *hidden caches*, and estimate their prevalence in the Tranco top 10k to be around 5%. Finally, we discussed a potential third methodology based on the `Date` header, which is only effective for specific cache technologies but not for others.

Collectively, these methodologies provide a comprehensive approach to detecting web caches, each with its own strengths and weaknesses. The Cache Header Heuristics is simple and effective when cache status headers are present, while timing analysis is more robust against stealthy caches but requires more requests. The `Date` header can be used as an additional indicator for some caches, but it is not universally applicable. These methodologies can be used together to improve the overall detection accuracy, allowing for a more thorough analysis of web caches across the Internet.

In the following chapters, we show how these methodologies can be exploited to detect web cache vulnerabilities in large-scale analyses of the Internet.

Chapter 4

How to Detect Web Cache Vulnerabilities

This chapter is based on works previously published as:

Mirheidari, S. A., Golinelli, M., Onarlioglu, K., Kirda, E., & Crispo, B. (2022). Web Cache Deception Escalates!. In 31st USENIX Security Symposium (USENIX Security 22) (pp. 179-196).

Golinelli, M., & Crispo, B. (2024, September). Hidden Web Caches Discovery. In Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses (pp. 65-76).

Golinelli, M., Arshad, E., Kashchuk, D., & Crispo, B. (2023, November). Mind the CORS. In 2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA) (pp. 213-221). IEEE.

In this chapter, we use the methodologies to detect caches and caching presented in the previous Chapter 3 to detect web cache vulnerabilities. We start by discussing the motivation for detecting web cache vulnerabilities. We then present our novel detection methodology for Web Cache Deception (WCD) vulnerabilities. Next, we use our methodology in a comparative experiment against the previous state-of-the-art methodology and in a large-scale investigation of the prevalence of WCD vulnerabilities in popular websites on the Internet. Finally, we present an investigation of cache poisoning Denial of Service (CPDoS) vulnerabilities introduced by Cross-Origin Resource Sharing (CORS) misconfigurations, which can be exploited to poison web caches. The vulnerabilities we focused on were selected based on gaps we identified in the literature:

understudied vulnerabilities with no automated methodologies to detect them.

Chapter Outline The chapter starts by discussing Web Cache Deception vulnerabilities. We present our novel detection methodology, a comparative evaluation against the previous state-of-the-art, and a large-scale experiment on the prevalence of WCD vulnerabilities in popular websites on the Internet. We then discuss Cache Poisoning Denial of Service vulnerabilities introduced by Cross-Origin Resource Sharing (CORS) misconfigurations, and present our methodology to detect and exploit these vulnerabilities.

4.1 Web Cache Deception

In this section, we present our novel methodology for detecting Web Cache Deception (WCD) vulnerabilities based on the Cache Headers Heuristics (CHH) presented in Section 3.2. We perform a comparative evaluation of our methodology against the previous state-of-the-art methodology, and a large-scale experiment on the prevalence of WCD vulnerabilities in popular websites on the Internet. We then tweak our methodology to detect WCD vulnerabilities using timing analysis on hidden web caches, i.e., caches that are not visible to clients since they do not send cache status headers.

4.1.1 Background and Related Works

In this thesis, we define Web Cache Deception (WCD) as the erroneous caching of dynamic content that is not cacheable when requested normally, but becomes cacheable when requested with a maliciously crafted URL. This definition does not make any assumptions about the impact of the attack; the erroneously cached content may or may not be valuable for an attacker. As long as caching happens contrary to the informed instructions of the website owner, an instance of a WCD vulnerability exists.

In their USENIX Security 2020 paper titled *"Cached and Confused: Web Cache Deception in the Wild"*, Mirheidari et al. presented the first study exploring WCD within a scientific framework [87]. In particular, they proposed a methodology for detecting WCD in the wild and conducted a large-scale study on 340 websites drawn from the Alexa Top 1k, finding 37 of them vulnerable. The authors also introduced novel WCD payloads, or *path confusion* techniques, and surveyed the top CDN vendors with their default caching configurations, highlighting the factors contributing to the

issue. This WCD detection methodology is highly relevant to our work, and we use the abbreviation CC to refer to it in the text.

At a high level, CC works as follows.

1. The tester creates an account on the website and populates user-editable fields that would normally hold personal or sensitive information with unique markers. Some examples of such fields are the email, the username, and the address. The fields are highly dependent on the type of website targeted.

2. A crawler with valid authentication cookies, collected after creating the account in the previous step, tests the pages of the website with WCD exploits. This crawler simulates a logged-in victim clicking on URLs containing WCD payloads.

3. A second crawler, this time without authenticating to the site, requests the same pages targeted in the previous step. This crawler simulates an attacker probing for successful exploits. If one of the responses contains a marker, the corresponding exploit from the previous step was successful in tricking a cache into storing the page, exposing the information to an unauthenticated request.

One advantage of this approach is its robustness against false positives; the presence of a marker is strong evidence that an information leak is taking place. In fact, Mirheidari et al. cite this property as one of the reasons they chose not to employ fuzzier detection techniques. On the downside, marker injection is a manual process. The authors also acknowledge this limitation, which forces them to cap their experiments at 340 websites, 295 of which are chosen specifically due to their support for Google OAuth, easing the account creation burden through automation support.

A more fundamental limitation of CC is that it is calibrated for WCD scenarios that involve leakage of personal information protected behind authentication gates. That comes at a cost: CC has no visibility into the caching behavior of a website when the page under test does not reflect user input (i.e., markers). In fact, some websites may not even have viable avenues for marker injection. Hence, CC forfeits the opportunity to detect vulnerabilities on such pages in order to achieve robust results on pages that do reflect user input. This is significant, because erroneous caching has implications beyond personal information leaks. Dynamic pages, be they publicly accessible or protected behind authentication gates, may include secrets such as CSRF tokens, CSP nonces, and OAuth state parameters, with dire consequences if stolen. Mirheidari et al. do allude to this possibility, but they are not equipped to explore that direction using CC.

4.1.2 Motivation

Our research is directly motivated by the limitations of prior work on WCD, and important gaps those may have left in the security community's understanding of WCD's spread and impact. We propose a new methodology DE, which challenges the core design decisions made for the state-of-the-art approach CC, and in doing so allows us to explore WCD in the wild at a depth and scale previously not possible. In doing so, we aim to equip website owners and researchers with better awareness, techniques, and tools to mitigate vulnerabilities, but also to estimate how easily miscreants can identify the same vulnerabilities.

In particular, we tackle the following limitations of CC.

(P1) **Coverage Problem.** CC cannot test web pages that do not reflect markers.

(P2) **Scalability Problem.** CC has the costly prerequisites of account creation, user input identification, and marker injection – all performed manually.

4.1.3 Methodology

Our new methodology DE uses a combination of content identicality checks and header inspection heuristics to overcome the limitations of CC. While the high-level approach is the same (i.e., launch a WCD attack, verify its success), DE may not be as intuitive as injecting and retrieving markers at a first glance. Therefore, we adopt a top-down presentation; we describe the high-level scheme first, and later dive into details.

High-Level Overview

Algorithm 2 presents the complete pseudocode for our approach. Given a URL to test for the presence of a WCD vulnerability, we perform checks in three steps. If all three checks pass, we conclude that the URL contains an exploitable WCD vulnerability. We explain these steps below.

Step 1 – Does the URL return dynamic content? The premise of WCD is tricking a cache into storing dynamically generated content, as static pages are unlikely to contain sensitive data. Therefore, as a first step, we request the input URL two times, each with a fresh client state, and compare the responses (lines 1-3). If the results are identical, we conclude that this is a static page, and we abort the test. Otherwise, the URL contains dynamic content, and we proceed.

Algorithm 2 DE testing an input URL for WCD.

```
   Input: URL
 1: result1 ← get(URL)
 2: result2 ← get(URL)
 3: if result1 ≠ result2 then
 4:     attackURL1 ← generateAttackURL(URL)
 5:     attackURL2 ← generateAttackURL(URL)
 6:     result1 ← get(attackURL1)
 7:     result2 ← get(attackURL2)
 8:     if result1 ≠ result2 and result1.cache = MISS then
 9:         result2 ← get(attackURL1)
10:         if result1 = result2 and result2.cache = HIT then
11:             return WCD detected
12:         end if
13:     end if
14: end if
```

Step 2 – When we launch a WCD attack, does the server still respond with dynamic content? The next step is launching a WCD attack by modifying the input URL with a WCD payload to craft an attack URL, and requesting it. The modification process is similar to the example we presented in Figure 2.1; we append a path component to the URL, which points to a non-existent style sheet. We randomize the file name to prevent Internet users from inadvertently accessing the same URL and getting poisoned cache contents. In our payloads we use the `.css` extension following the guidance from prior WCD literature; while the attack could work with other static file extensions, style sheets exist on virtually all websites, making them the optimal candidate for WCD tests.

We then make our WCD attempt by requesting this attack URL, simulating a victim visiting the link. One consideration here is to ensure that the server still responds with dynamic content to the request. That may not always be the case, for example, if the attack fails and the server responds with a generic error page. To tackle this problem, we generate *two* unique attack URLs with randomized payloads as described above (lines 4-5), launch two attacks by requesting both (lines 6-7), and compare the results (line 8, the first condition). If the results are identical, the attack has failed, and we abort the test. Otherwise, if the results differ, we proceed to the final step where we verify whether the attack was successful.

The avid reader may wonder why the dynamic content check in Step 1 is necessary if we perform a similar check again in Step 2. In a real-life test scenario, a website would be probed with multiple path confusion techniques, each resulting in a different

attack URL and exposing new WCD vulnerabilities. We use the 5 techniques presented in previous work [87], and propose 7 new ones later in our experiments. In other words, Step 2 would be repeated many times over, slowing down the tests and putting a heavy traffic load on websites. The check in Step 1 gives us an early opportunity to filter out static pages that are not of interest, using only one request pair – a significant optimization. We need to perform a second check in Step 2 for each WCD payload to ensure that the server still responds to the modified URL.

Step 3 – Is the origin response to the attack URL cacheable? Recall that for WCD to succeed, the origin server must serve a dynamic response that erroneously gets cached. Further breaking that down, on a vulnerable site, the attack URL we requested in Step 2 (i.e., simulating a victim interaction) must elicit a response from the origin server, but further requests for the same attack URL must be served from the cache (i.e., simulating how an attacker would retrieve the sensitive content).

In this final step, we precisely perform this check by inspecting the HTTP response returned when we first visited the attack URL (line 6), and the response for a repeat request for the same URL (line 9)[1]. Here, we use the *Cache Headers Heuristics* presented in Section 3.2. Specifically, we perform two sets of checks. First, we utilize HTTP response header heuristics to verify that the initial request was a cache miss (i.e., it was served by the origin), but the latter request was a cache hit (lines 8 and 10, both second conditions). Next, we compare the response bodies to verify that they are indeed identical (line 10, the first condition), which provides added assurance for the correctness of our header heuristics. If both checks pass, we conclude that the attack was successful, and that the URL has an exploitable WCD vulnerability.

Interpreting the Results

DE addresses both limitations of CC. We do not rely on the presence of a marker or any other particular reflected input on the page, therefore, DE can test any website for WCD, resolving the coverage problem (P1). Similarly, because there is no initial setup necessary, DE can run large-scale experiments on the Internet or complex private enterprise deployments, resolving the scalability issue (P2).

We achieve these properties by utilizing fuzzier detection techniques and heuristics. Heuristics can and do fail, presenting interesting trade-offs between DE and CC. Before we experimentally investigate these, we explain what our scheme is designed to detect,

[1]To verify the attack's success we could have used either of the two attack URLs we generated in Step 2. We chose to use the first one.

and the ways it can fail.

True Positives DE is designed to detect dynamic content that is *not cacheable* when requested through its normal URL, but is *erroneously cached* when requested with a maliciously crafted URL – the very definition of WCD. This definition does not make any assumptions about the *impact* of the attack; the erroneously cached content may or may not be valuable for an attacker. As long as caching happens contrary to the informed instructions of the website owner, an *exploitable* WCD vulnerability exists.

For example, some pages with non-sensitive content may include dynamic parts containing dates, server response time metrics, or email obfuscation strings. If these pages are normally not cacheable, but with a WCD attack they are cached, this is a true positive for our purposes, regardless of the value of the leaked content. The server and cache combination interacts hazardously, and a future update to the page with sensitive information would have a security impact.

False Positives Our definition of false positives directly follows from the above. Any finding that does not involve accidental caching of non-cacheable content is a false positive.

While this definition remains a constant, the particular *reasons* for false positive findings are closely tied to the WCD detection mechanism used. In CC, false positives are due to markers that a web application *intentionally* reflects in its responses. Even when there is no successful WCD attack taking place, the marker presence incorrectly signals to the crawler that sensitive information has leaked. Identifying such false positives requires a manual analysis of every finding and assessing whether the markers are returned due to WCD.

DE probes a page with a WCD payload, and checks whether the page is dynamic and whether it is cached. If both are true, it flags this as a finding. However, this detection mechanism cannot distinguish between *explicitly* and *erroneously* cached dynamic content.

Dynamic pages may still be explicitly configured to be cacheable by the website owner. In other words, the page would be cached even when requested normally, without a WCD attack. This may be due to aggressive server performance optimizations; for example, some non-sensitive dynamic objects could be allowed to be served from a cache, perhaps with a short TTL, even if they go stale. Alternatively, there could be human error; the website owner may have accidentally configured a dynamic page for caching – even though this is not an informed decision, it is still an explicit instruction.

Regardless of the circumstances, DE would incorrectly flag the situation as a successful WCD attack.

One advantage of DE over CC is that its false positives can be identified and removed automatically, without human analysis. This is a trivial check shown in Algorithm 3. Specifically, we take each URL DE flags as vulnerable, request it twice normally, *without using a WCD payload*, and use the same header heuristics to test whether the second response was served from the cache. A cache hit means that the URL is still cached when there is no attack, hence a false positive. This check can also be integrated into our methodology (Algorithm 2, lines 1-3) with no added traffic load.

Algorithm 3 Test if a DE finding is a false positive.

```
    Input: URL
1:  result ← get(URL)
2:  result ← get(URL)
3:  if result.cache = HIT then
4:      return False positive
5:  else
6:      return True positive
7:  end if
```

False Negatives DE relies on cache status headers to determine whether our WCD attempts indeed result in the prerequisite cache miss followed by a hit. Because cache status reporting mechanisms are not standardized, servers may return headers unknown to DE, or no headers at all. Furthermore, by design, DE does not authenticate to websites, and hence cannot test pages behind authentication gates. As a result, DE is bound to miss WCD vulnerabilities in the wild. The impact of false negatives is not trivial to quantify; there exists no ground truth. Thus, our results should be interpreted as a lower bound on vulnerabilities.

4.1.4 Comparative Evaluation

We now present the results of our first experiment, where we run both DE and CC on a dataset of 404 websites for a comparative evaluation.

DE with Authentication

In doing this exercise, we are primarily interested in understanding how our scheme compares to the marker injection approach; however, there is a confounding factor in

this experiment: DE cannot access pages behind authentication gates, whereas CC was specifically designed to test those pages *only*. Therefore, in order to investigate both the impact of the protocol change and authentication state on WCD detection efficacy, we introduce a third methodology, called $\mathsf{DE}_{\mathsf{auth}}$.

$\mathsf{DE}_{\mathsf{auth}}$ is a hybrid approach between DE and CC. It uses our novel detection scheme at its core, but like CC, requires an account to be manually created on the website so that the attack URL is requested (Algorithm 2, lines 6-7) with valid authentication cookies, simulating a logged in victim clicking on the malicious link. There are no other changes; $\mathsf{DE}_{\mathsf{auth}}$ probes the cache contents with an unauthenticated request like before, simulating an attacker (Algorithm 2, line 9).

4.1.5 The Experiment

We implement CC as described by Mirheidari et al. [87] and our two new schemes inside HTTP crawlers, and perform one crawl with each for a total of three runs. We set up our crawler to visit pages on any subdomain we may discover on the target website, and test at most 500 URLs on each FQDN.

We test each page with 12 attack URLs utilizing distinct WCD payloads. These include the original invalid path extension technique we illustrated in Figure 2.1, 4 path confusion techniques that exploit URL encoding discrepancies proposed by Mirheidari et al., and a further 7 novel encoding tricks we devise. We do not aim to position these new techniques as a scientific contribution; however, they are valuable for practical bug hunting situations. Readers can refer to Appendix A for examples and a breakdown of our findings for each.

We draw our crawl seed pool of 404 websites from the Alexa Top 100k. We choose these targets due to the marker injection requirements/limitations of CC, by following the general protocol described in "Cached and Confused". Specifically, we first crawl the front pages of Alexa Top 100k, and identify websites that support standard Single Sign-On schemes by searching for links containing keywords (e.g., login, register) and OAuth and OpenID Connect parameters. We then manually filter out websites that require sensitive credentials such as social security numbers or bank accounts for account creation. We end up with 404 websites, create accounts on them, inject markers into user-editable fields, and collect session cookies for each to be used by CC and $\mathsf{DE}_{\mathsf{auth}}$. This process necessarily yields a data set that carries the same biases as the one used in "Cached and Confused"; this is another limitation of CC, and it has no material impact on our comparative analysis.

Table 4.1: WCD detection performance, i.e., the number of websites flagged as vulnerable, for each methodology. Percentages are calculated over the entire crawl set of 404 sites.

	CC		DE_{auth}		DE		Combined	
Total Detections	21	(5.20%)	134	(33.17%)	129	(31.93%)	160	(39.60%)
True Positives	18	(4.46%)	115	(28.47%)	104	(25.74%)	123	(30.45%)
False Positives	3	(0.74%)	19	(4.70%)	25	(6.19%)	37	(9.16%)
Unique True Positives	2	(0.50%)	13	(3.22%)	5	(1.24%)	—	

Table 4.2: The number of vulnerable websites found to leak common categories of sensitive data by each methodology. There may be multiple leaks on a given website; columns do not add up to totals. Percentages are calculated over the total number of true positives for each methodology.

	CC		DE_{auth}		DE	
CSRF Token	4	(22.22%)	35	(30.43%)	39	(37.50%)
CSP Nonce	0	(0.00%)	1	(0.87%)	1	(0.96%)
OAuth State	0	(0.00%)	3	(2.61%)	2	(1.92%)
Session ID	2	(11.11%)	3	(2.61%)	3	(2.88%)
Personal Information	18	(100.00%)	16	(13.91%)	0	(0.00%)
Total Leaks						
Sensitive	18	(100.00%)	36	(31.30%)	39	(37.50%)
Potential		—	56	(48.70%)	50	(48.08%)
Harmless		—	23	(20.00%)	15	(14.42%)

We configure the DE and DE_{auth} crawlers to record the page differences during dynamic content checks for websites flagged as vulnerable, so that we can scan these with regular expressions to detect common categories of sensitive data that may be leaked by the attack.

In all of our experiments, we flag a tested site as vulnerable if it contains at least one URL impacted by WCD. We believe this is the most relevant metric for our purposes that also supports our research goals. In practice, our crawler often finds multiple vulnerable URLs on each target website. However, without an in-depth manual analysis of each finding, we cannot accurately determine whether these vulnerabilities truly stem from distinct caching configuration issues, or whether the different URLs in fact correspond to unique pages. This analysis is not feasible or essential for our research.

Results

Table 4.1 shows the results of our experiments with each methodology, where we detected a combined total of 123 websites vulnerable to WCD. Table 4.2 presents a break-

down of the leaked data we found on these sites.

True Positives The true positive findings confirm our hypothesis: markers severely limit WCD detection. Even though our dataset is specifically biased toward websites that *must* support marker injection, many otherwise vulnerable pages did not reflect those markers. In fact, CC could only test 244 (60.40%) of the websites, but the remaining did not have any pages with a marker present. As a result, CC identified only 18 vulnerable websites in our experiments, whereas $\mathsf{DE}_{\mathsf{auth}}$ and DE performed considerably better at 115 and 104 hits respectively.

$\mathsf{DE}_{\mathsf{auth}}$ had a slight edge over DE. As one might expect, the difference was due to the vulnerable pages behind authentication gates, which DE cannot access. For example, we manually confirmed that a vulnerable billing settings page on a target website was detected by $\mathsf{DE}_{\mathsf{auth}}$, but DE was redirected to a secure login page when testing the same URL.

Likewise, CC found 7 vulnerabilities that DE missed thanks to its access to authenticated pages; but, in addition, it caught 2 unique vulnerabilities that even $\mathsf{DE}_{\mathsf{auth}}$ missed. We verified that in one case this was due to the target website returning no cache status headers, defeating our new scheme. The other case appears to be a vulnerability that was fixed between our two experiment runs.

Finally, DE found 5 unique vulnerabilities that neither authenticated approach identified. We verified that these cases were due to the websites either explicitly sending cache control headers that prevent caching, or quietly ignoring all cache directives, when we attached a cookie to the request. As we discussed in Section 2, bypassing caching rules based on the presence of authentication cookies is a common option web caches provide to prevent hazardous caching. The unauthenticated DE scheme successfully defeated that protection.

False Positives Recall that the false positives of DE and $\mathsf{DE}_{\mathsf{auth}}$ can be eliminated automatically. However, we choose to present a clear breakdown of all false positives here to highlight the differences between CC and our new schemes. We apply our automated check to identify the false positives for DE and $\mathsf{DE}_{\mathsf{auth}}$, and perform a manual inspection of the context around the reflected markers for CC.

DE and $\mathsf{DE}_{\mathsf{auth}}$ both had higher false positives compared to CC. As discussed, this was due to their inability to distinguish between explicitly and erroneously cached dynamic content. While CC was more reliable in this department, some markers were indeed intentionally reflected in all responses from the web application as we previously

explained, and their presence did not imply WCD. For example, one website publicly listed its recent visitors, one of which was our marked username. CC falsely flagged this as a vulnerability.

Leaks To correctly interpret the data in Table 4.2, recall that a WCD vulnerability can only result in a damaging data leak if there is sensitive data on the page to begin with. In our analysis, we found that some vulnerable websites did not contain such data, and the dynamic content leaked in the cache was *harmless* (e.g., timestamps, email obfuscation strings). Other websites did contain seemingly-randomized values that may *potentially* be sensitive, but these did not match any patterns of common sensitive tokens. Unfortunately, we are not in a position to reason about this potentially-sensitive category without a white-box understanding of the impacted websites' backend logic. We reiterate that all cases still stem from exploitable, true positive WCD findings, albeit some without immediate consequences. We present a breakdown of these totals at the bottom section of Table 4.2. Also note that, for CC, detections are due to markers known to populate sensitive fields, and therefore all findings are sensitive by definition.

The top slice of Table 4.2 presents a breakdown of the leaks in the sensitive category, once again highlighting the differences between each approach. CC primarily detected personal information leaks, but a few other security tokens were present on the same vulnerable pages by happenstance. $\mathsf{DE}_{\mathsf{auth}}$ also detected 16 out of these 18 leaks without relying on markers, and myriad other sensitive leaks. DE performed similarly well for security tokens, but could not find personal information leaks without access to authenticated pages.

Summary

This experiment shows that the marker injection approach is limited by both its attack surface coverage and the variety of leaks it can detect. Overall, identicality and headers heuristics enable considerably better WCD detection. We also demonstrate that leaks of non-personal sensitive data with WCD are practicable. We investigate the implications of this finding in the upcoming sections.

That being said, the idea of using an authenticated crawling approach still holds merit. Both CC and $\mathsf{DE}_{\mathsf{auth}}$ perform well with detecting personal information leaks, whereas DE is inherently unsuitable for the task. Where the setup overhead is manageable (e.g., when penetration testing one's own environment), $\mathsf{DE}_{\mathsf{auth}}$ or perhaps a combination of all three approaches would expose the most vulnerabilities.

Table 4.3: The number of websites containing at least one WCD vulnerability, and websites that leak common categories of sensitive data. Percentages are calculated over the entire crawl set of 10K sites.

Vulnerable Sites	1188	(11,88%)
CSRF Token	436	(36.70%)
CSP Nonce	13	(1.09%)
OAuth State	34	(2.86%)
Session ID	63	(5.30%)

Nevertheless, DE remains the only viable option for a large-scale measurement, with its good detection performance and zero setup overhead. Equipped with this knowledge, we proceed with our experiment on the Alexa Top 10k. The findings in this section are already alarming, with 30.45% of our data set containing WCD vulnerabilities – well above the estimations in "Cached and Confused".

4.1.6 Large-Scale Experiment with DE

We now present our final experiment, where we run DE on the entire Alexa Top 10k, and describe concrete exploitation scenarios demonstrating real-life impact.

The Experiment

This experiment generally follows the previously established protocol, except for two important changes.

First, we enable the automated false positive filtering outlined in Algorithm 3, therefore eliminating all false positives in our results. *All numbers we report in this section represent true, exploitable WCD vulnerabilities.*

Second, we relax our definition of true positives by choosing not to test pages containing known harmless dynamic components. It is true that these pages may still be vulnerable to WCD, and while that may not be an immediate threat today, it may lead to a real-life exploit if the page is updated with sensitive content in the future. However, we opt to forgo testing these as a performance trade-off due to the limitations of our crawler resources and to minimize the traffic we generate. Specifically, during Step 1 of DE, we apply pattern matches on the dynamic components we find during identicality checks. If we detect a known email obfuscation mechanism, web analytics script, Edge Side Includes tag, timestamp, or error page that reflects our WCD payload, we conclude that the content is non-sensitive, and abort the test.

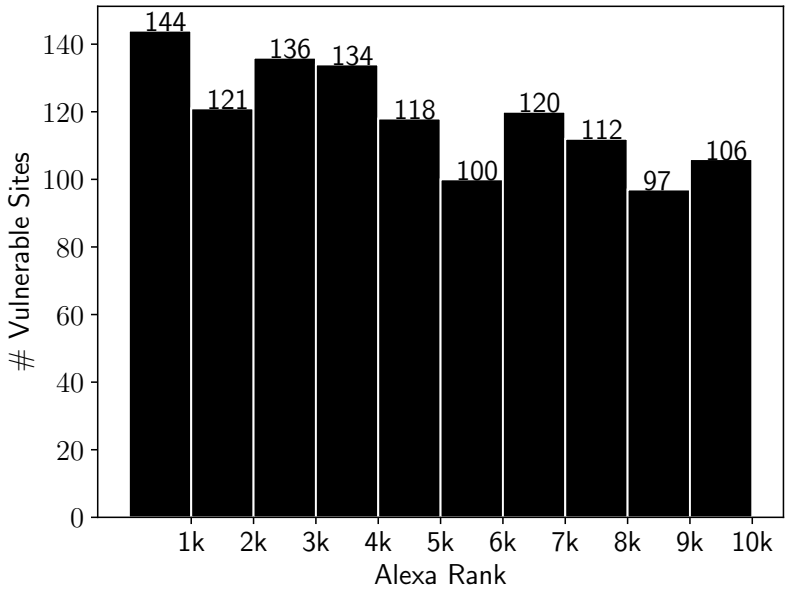

Figure 4.1: The distribution of vulnerable websites with respect to their Alexa ranking in 1K bins.

Results

Table 4.3 shows our findings. As a result of the aforementioned changes to the experiment protocol, we no longer need to report false positives or harmless data leaks – all flagged websites have true positive findings, and leak known or potentially sensitive values. We also do not have personal information leaks as DE cannot automatically detect them; however, we will demonstrate later that these findings assist us in finding personal information leaks upon further analysis.

1,188 websites among the Alexa Top 10k contain WCD vulnerabilities. This 11.88% incidence is significantly lower than the 30.45% we observed in the previous experiment; but we emphasize that the two results are not comparable. The previous dataset is non-uniformly drawn from the Alexa Top 100k based on the viability of marker injection; it is heavily biased. This larger dataset and the experiment have fundamentally different characteristics. Here, we study the most popular 10k websites likely to attract more attention from bounty hunters and attackers, and therefore discover and mitigate their vulnerabilities quickly. We also filter out the harmless leaks and report a looser lower-bound on vulnerabilities.

Figure 4.1 presents the distribution of vulnerable websites with respect to their Alexa ranks, exhibiting a fairly uniform, rectangular shape with a slight right skew. This suggests that WCD is pervasive among the websites in our dataset with no strong

connection to their popularity ranking.

4.1.7 Bounty Hunting with DE

All the WCD vulnerabilities we have reported in this work are *exploitable*, causing unintended content leaks into a public cache. However, a working exploit does not always equate to real-life *damage*; for instance, the vulnerable website may not process any sensitive data. Beyond the case studies we discussed above, we do not aim to measure such damage at scale in this work – that requires a manual analysis of each application and its data. However, we present a final empirical study to provide insights into the incidence of damaging exploits, and how vulnerable websites mitigate damage.

We perform this study on a separate dataset of 48 random vulnerable websites identified by running DE on domains listed on the bug bounty platforms HackerOne, Bugcrowd, Intigriti, and YesWeHack. This is not an arbitrary choice; obtaining the evidence we seek requires active exploitation of websites which provide a safe harbor for such testing in their infrastructure and reward bounties for damages that they acknowledge as real. We limit the scope by allowing DE to crawl a maximum of 50 pages on each website, and all manual analysis is performed by one researcher capped at a few hours of work. Therefore, readers should interpret our findings as the result of a best-effort attempt, but not a comprehensive penetration test.

Out of the 48 vulnerable websites, we were able to launch damaging attacks on **9**. These are similar to the case studies described above, and we omit their detailed discussion. 4 vendors paid out bounties, 2 acknowledged the issues but informed that another researcher reported it earlier, and the remaining 3 are still under evaluation.

Below is a breakdown of the reasons why we could not escalate the remaining WCD exploits to a damaging attack.

We were able to fully analyse the context around 24 websites, but there was no data valuable for an attacker. Another 10 websites did not allow us to explore the entire application, either disallowing public account creation, or requiring private information (e.g., a social security number) to proceed. We only analysed these partially, and found no valuable data.

3 websites leaked sensitive tokens, but this was not sufficient on its own. For example, a CSRF attack was stopped thanks to layered defences of referrer checks and captchas; a CSP nonce leak was useless as there was no XSS vulnerability to abuse it. 2 websites pulled sensitive data over an API at the browser side, therefore nothing damaging was cached.

This is decidedly a limited view into how WCD exploits escalate into end-to-end attacks. In an adversarial scenario, attacks may also be impeded by short cache eviction times, and cache locality in the case of distributed caches, as previously measured in "Cached and Confused". Regardless, we hope these added insights help qualify the core findings in our large-scale experiment. *Not every instance of WCD is an immediate threat; however, they are still exploitable vulnerabilities exposing applications to unpredictable risks.*

4.1.8 DE with Timing Analysis

We now present a variation of DE that can detect WCD vulnerabilities on pages that do not send cache status headers, i.e., hidden caches. As we showed in Section 3.3, this is a fairly common scenario in practice, and we detected 1,627 websites with such caches in our large-scale experiment on the Tranco top 50k.

Detection Methodology

Algorithm 4 Simplified pseudocode for our WCD detection methodology based on timing analysis. α is the significance level for the statistical test.

```
   Input: URL
 1: attackURL1 ← generateAttackURL(URL)
 2: attackURL2 ← generateAttackURL(URL)
 3: result1 ← get(attackURL1)
 4: result2 ← get(attackURL2)
 5: if result1 ≠ result2 then
 6:     timings1 ← [ ]
 7:     for i = 1 to n do
 8:         timings1.append(timingAnalysis(cacheBust(URL),
    cacheBust(URL)))
 9:     end for
10:     attackURL ← generateAttackURL(URL)
11:     timings2 ← [ ]
12:     for i = 1 to n do
13:         timings2.append(timingAnalysis(cacheBust(URL), attackURL))
14:     end for
15:     if t_test(timings1, timings2) ≤ α then
16:         return WCD detected
17:     end if
18: end if
```

Algorithm 4 presents a simplified pseudocode for our detection methodology. We

crawl each site to test and, for each URL that we visit, we test whether the response includes dynamic content or if the page is static. We do this by performing a simple string comparison of the responses. If the response is dynamic, we generate two attack URLs (lines 1-2), i.e., we modify the URL, including a path confusion payload and a WCD payload. A WCD payload comprises a non-existent file name and a static file extension (we use `.css`). Again, we check whether the response to the attack URLs is dynamic (lines 3-5). We perform this check because a WCD attack aims to induce a web cache into storing dynamic content that could contain sensitive data; a static file that is the same for all visitors is unlikely to include sensitive information. If the response to the attack URL is dynamic, we proceed with the timing analysis. The idea behind this methodology is to detect whether a response that includes dynamic content is cached and, therefore, if the website is vulnerable to WCD. We reiterate that our definition of WCD is the erroneous caching of dynamic content that is not cacheable when requested normally. Therefore, if we detect that a response with dynamic content is cached, we consider the website vulnerable to WCD.

Next, we send two groups of paired requests as follows:

1 `n` pairs of requests in a single packet to the base URL (i.e., the URL without added payloads), where both requests have random cache busters, i.e., the responses should always be served by the origin server (line 7-8).

2 `n` pairs of requests in a single packet where the first request has a random cache buster and the second request is to a single fixed attack URL (generated at line 10), i.e., the first response should be served by the origin server, the second by the cache *if the website is vulnerable to WCD* (lines 12-13).

We observe the order of arrival and measure the time elapsed between receiving the responses of paired requests and, using the t-test, we check if there is a significant difference in the timings between the two groups of paired requests (lines 15-16). If the difference is significant, we conclude that the response to the WCD-payloaded request is cached and, therefore, the website is vulnerable to Web Cache Deception.

Results and Case Studies

Of the 1.627 sites that our methodology detected as presenting a hidden cache, we detected that 1.020 (62.7%) are caching dynamic content. As black-box testers, we can't understand whether these websites are doing that deliberately, or if that indicates a Web

Cache Deception vulnerability. Testing all the affected websites for WCD vulnerabilities would be an extremely onerous manual task, and it is out of the scope of our research.

To validate our findings, we manually analysed a subset of these websites, limiting our tests to 35 randomly sampled sites among the ones caching dynamic content. In this analysis, our goal is to understand whether the identified vulnerabilities can be exploited to steal victims' sensitive information. Even if a site is caching dynamic information, this does not directly imply that the site is leaking private data, and the dynamic parts of web pages could be harmless content, such as timestamps and random error codes.

We registered test accounts on the sites that did not require personal information during the registration phase (e.g., a valid phone number, or a payment card), excluding the others. We successfully created a test account on 19 websites. Next, we checked how many of them cache dynamic content of pages also when the visitor is authenticated, resulting in 15 websites. Finally, we manually tested these websites to check if they leak sensitive information of authenticated users, finding that 5 websites are vulnerable to Web Cache Deception and leak private data.

Case Studies Our tests uncovered three large e-commerce websites vulnerable to WCD attacks. All of them leak personal information of the target victims, such as their email, geographical location and their shopping cart. Moreover, we discovered a large microblogging website vulnerable to WCD that leaked the emails of the target victims. It is important to specify that these vulnerabilities could not have been discovered using the state-of-the-art WCD detection techniques, due to the lack of cache status headers in the sites' HTTP responses. This highlights that our novel methodology is crucial for identifying well-hidden vulnerabilities that would otherwise be impossible to spot and detect.

4.1.9 Ethical Considerations

We now discuss the ethical considerations of our work, and how we mitigate the risks of negative security impact on the tested websites and their users.

No Harm to Users or the Internet

We carefully designed the methodologies and experiments in this paper to prevent a negative security impact on the tested websites or their users.

In particular, we never poison caches with malicious content, and never target Internet users with WCD. The personal information leaks explored in the paper are our own markers, and other sensitive tokens are the secrets that websites generate for our own test clients. In all case studies we play the role of the victim and attacker; we never target other users or launch exploits that persistently impact the target websites.

Furthermore, our path confusion techniques utilize randomized file names, meaning that cache keys corresponding to the erroneously cached content cannot feasibly be predicted or accidentally accessed by others. This is an added safeguard against confusing the websites' users. Even if the caches were accessible, there would be no danger to users; we never inject malicious payloads into the caches in the first place.

Coordinated Disclosure

We are committed to following coordinated disclosure procedures that exceed the established best practices. Unfortunately, with thousands of findings, especially those involving systematic issues that cannot be solved by deploying a common patch and therefore are out of scope for CERT assistance, this is not a straightforward process. The infeasibility of common approaches to large-scale vulnerability disclosures were documented in literature [122, 82, 121].

We adopted the guidance in the above literature to reach out to as many impacted parties as possible. We collected security contacts that were:

1 Disclosed on vulnerability management and bug bounty platforms. We collect contacts from HackerOne, Bugcrowd and Intigriti.

2 Compiled into open-source security lists.

3 Found in WHOIS records.

4 Published on the homepages of vulnerable websites.

For the websites we could not identify a security contact for, we emailed the generic inboxes *security@* and *privacy@*.

Table 4.4 shows a breakdown of the email addresses we collected via the different methods described above. In total, in both experiments, we identified 1279 vulnerable websites, 420 of which were only vulnerable due to a supply chain issue, for which we contacted the service provider and not all single impacted websites. Collectively, we contacted 859 websites.

Table 4.4: Number of contact points collected using different methods.

Platforms	Open-source lists	WHOIS records	Homepages	Generic emails	Total
18	76	54	182	529	859

These exhaust the viable options available to us. The cases that may not be covered by the above require deep exploration of the website or filling out non-automatable forms, which we could only do on a best-effort basis.

We began notifications promptly after finalizing the experiments, and gave website owners over 3 months to implement mitigations before a public disclosure. Our notification emails included our affiliation, a summary of WCD and our experiments, and a report of the findings pertinent to each party.

4.1.10 Mitigations

While there is no standard path to mitigate Web Cache Deception vulnerabilities, here we discuss some general strategies that website owners can adopt to protect their users. Together, these strategies form a multi-layered defence approach that can significantly reduce the risk of WCD attacks.

Bypass the Cache when Cookies are Present A common practice to prevent hazardous caching is to configure caches to bypass caching when authentication cookies (or other forms of session identifiers) are present in the request. This is effective against WCD attacks that target authenticated pages, but it does not protect public pages that may still contain sensitive data, such as security tokens. While this approach is easy to implement and effective in certain scenarios, it does not tackle the root cause of the problem. For these reasons, we recommend this approach only as one layer of a multi-layered defence strategy.

Use Appropriate Cache-Control Headers The ideal way to prevent WCD vulnerabilities is to ensure that dynamic content is never cached in the first place. Website owners should carefully audit their caching infrastructure and ensure that all dynamic content is served with appropriate cache-control headers and that the cache is configured to respect these headers. Such headers include `Cache-Control: no-store` or `Cache-Control: private`, to prevent caching by shared caches, as specified in RFC 9111 [42]. This approach directly addresses the root cause of WCD vulnerabilities and

is effective in all scenarios. However, we have frequently observed reticence from website owners to implement this solution due to backward compatibility concerns. Many web applications have complex infrastructure and frequently rely on legacy components that may not handle cache-control headers correctly or may be hard to update and reconfigure. Therefore, we recommend this approach as the primary mitigation strategy, but recognize that it may not be feasible for all websites.

Validate the Content-Type A key aspect of WCD attacks is the mismatch between the expected content type of a resource based on the extension in the URL and the actual content type of the response. When caching decisions are solely based on the URL, this mismatch can lead to dynamic content being cached under the guise of a static resource. To mitigate this risk, website owners should implement validation mechanisms that ensure the content type of a response matches the expected type based on the URL. For example, if a URL ends with `.css`, the server should verify that the response has a `Content-Type: text/css` header before allowing it to be cached. If there is a mismatch, the server should either reject the request or serve an appropriate error response. This approach adds an additional layer of security by ensuring that only content of the expected type is cached, thereby reducing the risk of WCD vulnerabilities. This strategy is the solution proposed by Cloudflare, implemented in their CDN as a configuration option that is, however, disabled by default [24]. This mitigation is effective and relatively easy to implement, while also being highly compatible with existing web applications and caching infrastructures. Therefore, we recommend this approach as a complementary mitigation strategy alongside the others discussed above.

Regularly Audit Caching Configurations All the mitigation strategies discussed above require careful configuration of the caching infrastructure. Website owners should regularly audit their caching configurations to ensure that they are correctly implemented and effective against WCD attacks. This includes testing for WCD vulnerabilities using tools like DE, as well as reviewing cache-control headers. Regular audits help identify potential misconfigurations or vulnerabilities that may have been introduced over time, allowing website owners to take corrective actions promptly. This proactive approach is essential for maintaining the security of web applications in the face of evolving threats. Our tools are publicly available to facilitate such audits.

4.1.11 Section Summary

We directly tackled the limitations of the state-of-the-art approach in WCD vulnerability detection, subsequently conducting the largest-scale WCD measurement over 10k websites. Here, we summarize our research findings.

Through our comparative experiment, we demonstrated that our new methodology DE addresses both the coverage (P1) and scalability problem (P2), and it can indeed significantly outperform CC. However, we also showed that CC and the authenticated variation of our scheme, $\mathsf{DE}_{\mathsf{auth}}$, open up opportunities to identify additional vulnerabilities. Where scalability is not a concern, a combination approach is ideal.

We showed with our large-scale experiment that over 4 years after the conception of the attack, and 2 years after the experiments in "Cached and Confused," WCD is still distressingly pervasive. This aligns with the popularity of the attack on bug bounty platforms – and likely miscreant activity that goes unnoticed.

Our experiments and case studies illustrated that there is an abundance of sensitive security tokens present on publicly accessible pages, which can be stolen via WCD to bypass standard defences and facilitate real-life attacks. Many websites that leak such tokens are evidently impacted by WCD in more than one way, exposing flaws that lead to further attacks and leaks. These observations, combined with the significant performance advantage of DE over CC, suggest that focusing on personal information sources and sinks for WCD detection is not the most effective detection strategy, even when testing individual websites in a controlled setting.

We adapted our methodology DE to use timing analysis to detect WCD vulnerabilities in the hidden caches identified during our large-scale experiment on the Tranco Top 50k presented in Section 3.3, finding that 1.020 websites out of 1.627 (62.7%) cache dynamic responses. Caching dynamic responses is not necessarily an indication of an exploitable vulnerability, but it certainly is a hazardous behaviour that, in specific circumstances, might lead to the leakage of sensitive information of the websites' visitors. We manually investigated 35 randomly sampled websites that cache dynamic data and identified 5 websites vulnerable to WCD attacks that could be exploited to leak sensitive information about victims. We find that hidden caches are affected by common cache vulnerabilities and a vast majority of them cache content that they should not. This methodology is crucial to identify well-hidden vulnerabilities that would otherwise be impossible to spot.

With this research, we contribute novel insights into the scale and impact of the problem. The methodology we present will help website owners test their own systems

for vulnerabilities, and researchers to run experiments with ambitious scopes. However, another implication of this work is that attackers, too, can quickly identify vulnerabilities en masse. WCD, and web cache attacks in general, require immediate attention from the security community for a robust solution.

Discussion

Before we conclude, we reiterate that WCD is a system problem. Individual components such as the clients, web servers, proxy services, or CDN providers are not necessarily faulty in isolation; their complex interactions give rise to unexpected and dangerous caching decisions. One corollary of these circumstances is that our findings do not implicate the developers and operators of these individual components. But, perhaps the more critical take away is that website owners cannot rely on traditional vulnerability management and software testing processes to eradicate these vulnerabilities – there is often no unit test to run, no signature to check, no CVE to track, and no patch to deploy. It is not yet clear whether mapping complex traffic flows and analysing them holistically for cache attacks is feasible, or even possible. That remains an open challenge for the security research community, and in light of the resurging popularity of web cache attacks, we believe it has already become a pressing line of investigation.

In the meantime, our work presents one key takeaway for website owners who are inevitably getting more familiar with the escalating web cache attacks: CDNs and caching proxies are powerful technologies in an already complex ecosystem. Simple caching rules can have far-reaching effects, and making assumptions about the cacheability of objects based on their public exposure to the Internet alone is, evidently, *unsafe*. Website owners should carefully consider (and test) the security implications of changes to their caching infrastructure, and exercise caution when using blanket rules such as those that cache all objects served from a given endpoint or all files with a given extension.

4.2 Cache Poisoning

In this section, we show how implementation flaws and errors in Origin validation for the Cross-Origin Resource Sharing (CORS) security mechanism enable attackers to perform Denial of Service through cache poisoning. Specifically, we first detect CORS misconfigurations that mistakenly trust unintended origins, and then exploit misconfigured web caches to poison cached CORS responses, resulting in Denial of Service for legitimate users.

4.2.1 Background and Related Works

We start by presenting an overview of how Cross-Origin Resource Sharing (CORS) works and discuss related concepts such as Same-Origin Policy (SOP), cross-site requests, and describe the risks involved in cross-origin communications.

Cross-Origin Resource Sharing

The Same-Origin Policy (SOP) is a fundamental concept in web application security. The SOP sets access restrictions on web resources, isolating them and providing boundaries from other websites with different origins. The origin of a website is defined as the combination of its protocol, host, and port. Websites are considered from the same origin if and only if all these three values are exactly the same [111].

The SOP quickly became too restrictive for a large portion of websites that need to communicate beyond the boundary of their current origin to provide their services. As the web becomes more complex, more websites rely on exchanging data with other websites to provide better functionality to their users. To address this problem, in 2006, the W3C introduced a mechanism, called Cross-Origin Resource Sharing (CORS), to relax the SOP allowing cross-origin requests and sharing of resources between websites with different origins [18].

CORS is an extension of the `XMLHttpRequest` API and functions through a set of HTTP headers enforced by the client to provide permission to access the selected resources in cross-origin requests. The server performs the validation, authorization, and access restrictions, while the browser is responsible to support these HTTP headers to enforce the restrictions. This protocol has been adopted by all major browsers, and has been widely adopted by websites.

We take websites A and B as an example to describe CORS in a real scenario. Website A requests a resource from website B and the SOP will not grant this access unless B uses CORS and responds with an "Access-Control-Allow-Origin" (ACAO) header, indicating A's origin as a trusted party to access that resource. To check whether the server is configured to use CORS, the browser sends a preflight request, i.e., an OPTIONS HTTP request that includes the three headers: "Access-Control-Request-Method", to discover which methods are supported, "Access-Control-Request-Headers", and "Origin" [138]. ACAO header supports a single origin within a CORS response. To allow any origin, the server sets the header a wildcard `*`. An additional header, "Access-Control-Allow-Credentials" (ACAC), allows credentialed requests when set to true. A credentialed request includes authentication mechanisms, such as cookies and

authorization headers. Credentialed requests with a wildcard are forbidden [86].

Cache Poisoning Denial of Service

If a website uses CORS, web pages are cached, and the value of the Origin header is not included in the cache key, an attacker can perform a Denial of Service (DoS) attack enabled by CORS flaws. The attacker sends a request with a maliciously crafted Origin header that is mistakenly trusted by the web server. The server responds with an ACAO header that includes the malformed Origin value, and the response is cached. Subsequent requests for the same resource with a different Origin header will receive the cached response, which includes the malformed Origin value. When the browser receives this response, it checks whether the Origin header of the request matches the ACAO header of the response and, since they do not match, blocks the response, resulting in a DoS for that resource.

Related Works

Several academic and non-academic researchers have identified security problems with CORS, highlighting common CORS flaws and their implications. Kettle [72] provides a summary of CORS flaws identified throughout his penetration testing experiences. Müller [90] takes different CORS flaws and measures their prevalence on the Alexa top 1M, while Evan J performs a measurement of the arbitrary origin reflection in [62], showing a high number of vulnerable websites in the Alexa top 1M. Chen et al. [18] provide an empirical study of CORS security, finding new issues in the design and implementation of CORS that lead to Remote Code Execution (RCE), file upload CSRF and attacks on binary protocol services [18]. Meiser et al. create a graph of interconnected trust relationships between websites, considering existing cross-communication methods such as `postMessage`, CORS and domain relaxation. They specifically focus on the dangers that the interconnected network of trust could cause and investigate the attack surface [86].

4.2.2 CORS Flaws

Web developers frequently generate dynamic CORS policies to dynamically validate the value of the request's "Origin" header at runtime. If these policies are not implemented correctly on the server side, the web application might unintentionally trust domains that it is not intended to trust, hindering the enforcement of the SOP. Due to the

complexity of this validation process and the pitfalls in CORS design, several types of flaws can arise in its implementations. Following we list the types of CORS flaws that we investigated in our research. Flaws are compiled based on previous research, and are supplemented with novel flaws that we present here [18].

Arbitrary origin reflection Servers blindly reflect the value of the "Origin" request header in the ACAO response header. This implementation effectively trusts any domain that sends the request.

Prefix matching The server trusts any domain prefixed with a trusted domain. For instance, a server mistakenly allows "victim.com.attacker.com" when it wants to trust "victim.com".

Suffix matching The server only checks if the origin ends with the trusted domain. This is a common practice to allow all the subdomains of a domain. For example, if a server wants to allow "victim.com", it mistakenly trusts "attackervictim.com" too.

Not escaping '.' When the validation is performed using a regular expression, the developer might neglect escaping dot characters ., used as a wildcard in regular expressions. For instance, "victim.com" wants to allow "www.victim.com", but allows "wwwAvictim.com" as well.

"null" value The "Origin" header was first proposed as a mitigation against CSRF attack, and CORS reuses this header [8]. One of the essential conditions for CORS security is that the "Origin" header value cannot be forged in a cross-origin request, but this is not always true in reality. RFC 6454 states that a request from a privacy-sensitive context should set the "Origin" header to "null", even though it does not provide an explicit definition for privacy-sensitive contexts [8]. Moreover, CORS standards do not define the "null" value clearly. In reality, browsers send the "null" value in several scenarios, such as from iframe sandboxed scripts. To share data with these types of sources, a developer must allow the null value in their configuration, setting "Access-Control-Allow-Origin: null" and "Access-Control-Allow-Credentials: true". By doing so, an attacker can easily forge the "Origin" header by sending a cross-origin request from an iframe sandbox in the browser. Consequently, websites with this configuration can be accessed by any other websites.

HTTPS site trusts HTTP domain Some CORS configurations do not take the scheme into account and cause HTTPS sites to trust HTTP ones.

Arbitrary subdomains Most applications decide to allow access from all their subdomains, even non-existent ones.

End matching This is different from suffix matching in that the website is not only checking for a trusted domain, but also for the scheme. Consequently, a website that wants to trust "https://victim.com" trusts "https://attacker.com/https://victim.com" by mistake. This flaw cannot be exploited by a web attacker because the browser will never generate such a value for the Origin header; however, this variation can be exploited to achieve DoS by manually crafting an HTTP request with a malformed Origin.

4.2.3 Methodology

In this section, we present our methodology to detect cache poisoning Denial of Service vulnerabilities due to CORS flaws. We first describe how we test URLs for CORS flaws, and then how we check if they are vulnerable to cache poisoning attacks.

4.2.4 Testing URLs for CORS Flaws

When testing a URL for CORS flaws, we must first check whether the URL is configured to use CORS. If it is not, we can skip the test. To do this, we send a request to the URL with the "Origin" header with a genuine value and check the presence of the ACAO header in the response. If the ACAO header is present, the URL is configured to use CORS, and we can test it for the previously described CORS flaws. If the ACAO header is not present, we can conclude that the URL is not configured to use CORS, and we can skip the test. We developed a tool that we call *CORS Flaws Scanner* that sends an HTTP request for each flaw to each web page, mutating the value of the Origin header accordingly to the variation. If the mutated Origin value is reflected in the response ACAO header, the web page is flawed for the tested variation.

4.2.5 Testing URLs for Cache Poisoning Vulnerabilities

As explained in the background section, this attack is possible when the website presents a CORS flaw, the target web page gets cached, and the Origin header is not keyed by

Table 4.5: Experiment statistics: number of sites and pages that we visited that use CORS and are vulnerable to at least one CORS flaw. Percentages for each row are calculated over the number of visited sites or pages in the *Visited* column.

	Visited	Use CORS	Vulnerable
Websites	39,067	6,862 (17.6%)	2,014 (5.2%)
Total web pages	58,815	7,681 (13.1%)	2,350 (4.0%)
Homepages	39,064	5,328 (13.6%)	1,536 (3.9%)
Login pages	19,751	2,353 (11.9%)	814 (4.1%)

the web cache. To check for sites that present these characteristics and are therefore vulnerable to DoS we use the following methodology. We first send a request with a maliciously crafted Origin that is mistakenly trusted by the website, so that its value is reflected in the ACAO response header. Next, we send a second request, this time with a genuine Origin based on the URL of the web page, and check whether the value of the response matches the genuine Origin or the maliciously crafted one. If the ACAO value of the second response includes the modified Origin value, the website is vulnerable to cache poisoning and prone to DoS attacks. To avoid poisoning actual resources that could be accessed by genuine users of the tested websites, we cache-bust our requests by adding random query parameters. This also ensures that the responses are not already cached when we start our tests.

4.2.6 Experiment

To quantify the prevalence of CORS flaws and cache poisoning vulnerabilities, we run our tool on the Tranco top 50k list generated on April 4, 2023 [99]. [2] For each website, we test for CORS flaws the homepage and the login page, if we can identify it. We choose the homepage and the login page to have data on a resource that can be visited with authentication (by logging into the sites) and one that should only be accessible by unauthenticated visitors. This also allows us to assess the impact of CORS flaws against unauthenticated victims. In total, we identified 39,064 homepages and 19,751 login pages on a total of 39,067 websites. For 10,933 websites, we could not identify the homepage or login page due to various problems. Primarily, we encountered errors such as request time-outs, hosts not resolvable by the DNS, and unreachable destinations. Additionally, some domains lacked an actual homepage (e.g., domains that serve static resources and answer HTTP requests only for specific paths). In total, we successfully

[2] Available at https://tranco-list.eu/list/W9J39

Table 4.6: Number of pages and sites misconfigured to the tested variations. Percentages are calculated over the total number of flawed pages or sites, presented in the last row of the table for each column.

ID	Variation	Homepages	Login pages	Sites
1	**HTTPS trusts HTTP**	1391 (90.6%)	682 (83.8%)	1786 (88.7%)
2	**Arbitrary subdomain**	606 (39.5%)	396 (48.6%)	848 (42.1%)
3	**Arbitrary origin reflection**	484 (31.5%)	248 (30.5%)	655 (32.5%)
4	**"null" value**	476 (31.0%)	240 (29.5%)	640 (31.8%)
5	**Prefix matching**	110 (7.2%)	65 (8.0%)	159 (7.9%)
6	**Non escaped dot**	123 (8.0%)	45 (5.5%)	146 (7.2%)
7	**Suffix matching**	85 (5.5%)	24 (2.9%)	91 (4.5%)
8	**End matching**	307 (20.0%)	177 (21.7%)	422 (21.0%)
	Total	**1536**	**814**	**2014**

tested 58,815 pages, 7,681 of which (on 6,862 different websites) responded with CORS headers.

We test the collected web pages using the *CORS Flaws Scanner*. As seen in Table 4.5, we found 2,014 sites misconfigured to at least one of the variations we tested for, with a total of 2,350 misconfigured web pages. Table 4.6 shows the number of sites and pages misconfigured to specific variations. When counting, we only consider the most generic variation, disregarding the most specific ones. For example, if a website is misconfigured to "arbitrary origin reflection", we only count it for that, even though all the other variations would test positive.

Next, we check whether the misconfigured sites are vulnerable to cache poisoning attacks. We identified 45 sites that are vulnerable to this attack. An attacker can abuse any variation, since it is enough for the injected value to differ from the Origin value of genuine HTTP requests for the attack to succeed. While this number may seem small, it is important to note that we only a limited set of web pages.

4.2.7 Ethical Considerations

During the execution of the attacks that we reproduced and described, we always used specifically created test accounts as victims. No users of any website were impacted by our attacks. We employed cache-busting techniques to avoid poisoning resources potentially accessed by genuine users of the target websites. During our experiments, we

minimized the impact on targeted websites by limiting the number of requests performed to the minimum possible; moreover, we performed no disruptive attacks. We have never disclosed the domains of sites impacted by these vulnerabilities to any third party and in this article.

4.2.8 Mitigations

To mitigate the risk of cache poisoning Denial of Service attacks enabled by CORS flaws, we recommend website owners to implement the following strategies.

Correctly Implement CORS Policies An effective way to prevent cache poisoning DoS attacks is to ensure that CORS policies are correctly implemented on the server side. This includes preferably avoiding dynamic CORS policies that validate the Origin header at runtime, and instead using static policies that explicitly allow only trusted origins. If dynamic policies are necessary, website owners should ensure that the validation logic is robust and does not inadvertently trust unintended origins. This can be achieved by thoroughly testing the CORS implementation for common flaws, such as those described in this section. Our tool can be used to automate this testing process. However, relying solely on testing is not sufficient to ensure security.

Include the Origin Header in Cache Keys The most direct mitigation against cache poisoning DoS attacks is to include the value of the Origin header in the cache key. By doing so, the cache will store separate responses for each unique Origin value, preventing an attacker from poisoning the cache with a maliciously crafted Origin. This approach effectively isolates cached responses based on their Origin, ensuring that a response intended for one origin cannot be served to requests from another origin. Implementing this strategy requires configuring the caching infrastructure to recognize and incorporate the Origin header into its caching logic. This should not be considered a mere security best practice: it is an essential requirement for any website using CORS to prevent cache poisoning attacks.

4.2.9 Section Summary

29.4% of the websites using CORS had at least one implementation flaw. Specifically, the vast majority of these sites trust the HTTP version of their domain; nearly half allow arbitrary subdomains and one in three trusts the value null or reflects the value of the request origin in the response. These results suggest that developers are either unaware

of the potential security consequences of these dangerous behaviours or underestimate them. We believe that devolving the burden of validating the Origin programmatically on the server, instead of introducing a policy-based system enforced directly by the browser, has introduced an inherent complexity to the CORS mechanism, which is thus prone to human error with severe consequences for web security. Our investigation also discovered 45 sites vulnerable to DoS attacks enabled by CORS flaws and cache poisoning, as they fail to include the Origin value in the cache key.

4.3 Chapter Summary

In this chapter, we presented our novel methodologies to detect Web Cache Deception vulnerabilities, and cache poisoning Denial of Service attacks enabled by CORS flaws. We first introduced DE, a novel methodology that can detect WCD vulnerabilities in a scalable way, and showed that it can outperform the state-of-the-art CC in terms of coverage and scalability. We then presented a large-scale experiment on the Tranco top 50k websites, where we found that WCD is still prevalent on the web, with 10.2% of the tested websites vulnerable to WCD. These vulnerabilities can be exploited to steal sensitive information and security tokens of victims. Finally, we adapted our methodology to detect WCD vulnerabilities in hidden caches, finding that 62.7% of them cache dynamic responses, which could lead to the leakage of sensitive information of victims. Next, we investigate how CORS flaws can lead to Denial of Service, finding that 45 sites in our dataset fail to include the Origin value in the cache key and are therefore affected.

In the next chapter, we show how these vulnerabilities can be chained with other attack vectors to create sophisticated attacks, and how they can be exploited to bypass security mechanisms and protections.

Chapter 5

Security Impact and Case Studies of Web Cache Vulnerabilities

This chapter is based on works previously published as:

Mirheidari, S. A., Golinelli, M., Onarlioglu, K., Kirda, E., & Crispo, B. (2022). Web Cache Deception Escalates!. In 31st USENIX Security Symposium (USENIX Security 22) (pp. 179-196).

Golinelli, M., Bonomi, F., & Crispo, B. (2023, September). The Nonce-nce of Web Security: An Investigation of CSP Nonces Reuse. In European Symposium on Research in Computer Security (pp. 459-475).

In this chapter, we explore the broader security implications of web cache vulnerabilities, particularly focusing on Web Cache Deception (WCD). While previous chapters have primarily concentrated on the detection and prevalence of WCD vulnerabilities, this chapter explores the diverse and often unexpected consequences that arise from these vulnerabilities. We present real-world case studies that illustrate how WCD can be exploited in various contexts, leading to significant security breaches beyond the immediate data leaks typically associated with such vulnerabilities. Furthermore, we investigate the reuse of Content Security Policy (CSP) nonces, a critical security mechanism designed to prevent Cross-Site Scripting (XSS) attacks, and how their improper caching can undermine web security. Through these discussions, we aim to highlight the multifaceted nature of web cache vulnerabilities and their potential to facilitate complex attack vectors.

Chapter Outline The chapter is structured as follows. We begin by discussing the security impact of WCD vulnerabilities and presenting real-life case studies that illustrate their potential for abuse. We specifically highlight scenarios where WCD vulnerabilities are chained with other vulnerabilities to create sophisticated attacks. Next, we explore the reuse of CSP nonces and how their improper caching can undermine web security. Finally, we conclude with a discussion on the broader implications of web cache vulnerabilities and potential mitigation strategies.

5.1 Security Impact of WCD and Case Studies

In the previous chapters, we have demonstrated how to detect WCD vulnerabilities and measured their prevalence on popular websites. We now discuss the security implications of these vulnerabilities, which are not only limited to the immediate data leaks they cause. In this section, we explore the broader security impact of WCD and present real-life case studies that illustrate their potential for abuse.

Our findings already imply that the leaked sensitive tokens may be abused by an attacker to break the security mechanisms each support. For instance, leaked CSRF tokens enable confused deputy attacks, CSP nonces break defences against inline JavaScript inclusions, and OAuth state parameters and session IDs enable hijacking victim accounts or stealthily logging victims into attacker-controlled accounts.

However, the implications of our findings extend beyond these basic attacks. In this section, we present real-life case studies drawn from our experiment, and provide insights into the less obvious damage potential of WCD. These discussion points demonstrate that WCD has ramifications distinct from personal information leaks.

Due to the excessive number of vulnerabilities we identified, it is not feasible to investigate all findings systematically. The below scenarios represent an arbitrary list of real-world attacks that nevertheless demonstrate the severity of WCD. We chose these particular targets for manual exploration motivated by the website owners' presence on vulnerability management platforms, so that we could rapidly communicate and help mitigate any issues. All attacks described below were carried out with a test user, no actual Internet users were targeted or harmed.

5.1.1 Leaked Tokens Lead to Standard Attacks

We first describe two representative attacks made possible by stealing the sensitive tokens listed in Table 4.3 via WCD to give readers assurance that the impact is practical.

We found a popular travel and lodging reservation platform to leak session IDs. We were successfully able to use this stolen token to hijack customer service chat sessions of an unauthenticated user. The same attack translated to authenticated users as well; when a logged-in user visited the WCD exploit link, we were able to hijack their entire session and access complete booking details.

In another instance, we identified that the error pages on Mozilla Thunderbird's add-ons portal were vulnerable, and they contained registration and login links with OAuth state parameters. By stealing this value we launched a *Login CSRF* attack [124], which allowed us to trick a victim into unknowingly logging into an account we controlled, hence enabling us to view their activity and the information they enter. Mozilla fixed the issue within 24 hours of our notification.

These attacks demonstrate that sensitive token leaks on publicly accessible pages pose a real threat to unauthenticated visitors of a website as well as logged-in users. As an additional empirical observation, a plethora of other traditional CSRF and session hijacking attacks were possible via WCD, but we noticed that damage was sometimes contained thanks to layered defences such as referrer checks and captchas. This once again highlights the importance of a defence-in-depth strategy for practical web security.

5.1.2 WCD Leads to Cache Poisoning

WCD is a specialized subcategory of cache poisoning attacks, where a cache is tricked into storing and leaking sensitive data. That being said, the underlying mechanism for exploitation remains the same for all such cache attacks: content is erroneously cached. This implies that the vulnerable websites we detected may be exposed to other varieties of cache attacks, regardless of whether they immediately leak any sensitive data.

We found one such instance to impact a major American payment processor. Many pages on this website were impacted by a *reflected* cross-site scripting (XSS) vulnerability, where the value of the `X-Forwarded-Host` header included in requests was printed on the page without output sanitization. This enabled arbitrary script injection attacks. However, since attackers cannot control the headers sent by the browser of a victim, this attack is considered a *self-XSS* attack, which is generally not considered a real vulnerability and is out of scope for responsible disclosure policies. Moreover, as with many reflected XSS attacks, the avenues for exploitation would normally be limited. However, this website was also vulnerable to WCD. An attacker could combine the two vulnerabilities, and consequently cause the fronting cache to store the response together with the reflected XSS payload. This escalates the attack to a *stored* XSS, where the

injected malicious payload is now automatically served from the cache to unsuspecting clients visiting the website.

This attack illustrates that WCD has dire consequences even when the website has no sensitive data to leak. Identifying such caching hazards is key to preventing complex, non-obvious system issues that may be lying dormant.

5.1.3 Token Leaks Correlate to Personal Information Leaks

DE is not designed to catch personal information leaks. However, our manual analysis shows that the presence of a WCD vulnerability on a public page is often indicative of more WCD issues that impact pages protected behind authentication gates, and therefore endanger personal information, too.

While we cannot scientifically quantify the incidence or reasons without a dedicated study, one intuitive explanation is that there is no fundamental difference between caching misconfigurations that lead to WCD vulnerabilities affecting authenticated and unauthenticated victims. Thus, a caching rule that leads to erroneous content storage on a public page may enable the same attack on a protected page in the absence of a session or cookie-based cache bypass mechanism.

We selected 55 websites flagged by DE that support user accounts, implying that they contain personal information. We created test accounts on these websites, and attempted WCD attacks on pages that require authentication for access. In 10 out of 55 cases, we were successfully able to cause personal information fields to get cached. To provide insights into the type of information that could be leaked, these were well-known websites including a domain registrar, a travel reservation platform, a job application and company review portal, an online course provider, a security product vendor, and a cryptocurrency exchange.

While this is not conclusive evidence, 18% is a non-negligible success rate. This suggests that our approach of detecting WCD vulnerabilities by performing checks on publicly accessible pages do not completely forfeit the opportunity to detect personal information leaks. Website owners should carefully examine vulnerabilities lest they remain exploitable in different authentication contexts.

5.1.4 WCD Poses a Supply Chain Issue

Recently, highly-publicized cybercrime campaigns such as the Magecart attacks [123] and the SolarWinds incident [123] have put a spotlight on supply chain attacks, alerting the security community to the widespread damage one vulnerable supplier or vendor

may cause to the Internet ecosystem. In our experiment, we found that supply chain attacks are not limited to the traditional malicious code inclusion vectors, and that a single vulnerable online service provider with a caching hazard can expose many websites to WCD.

We identified a multitude of vulnerable URLs in our results that share an identical subdomain and similar path components (i.e., *support*.example.com/*common-pattern*). Upon manual inspection, we determined these pages to be integration points with a popular customer service and support management platform. Due to the WCD vulnerabilities present on this vendor's platform, many (or, potentially all) of their customers were also impacted under their respective domains. To demonstrate the weight of the issue, 399 out of the 1188 websites we flagged were expressly due to this vulnerability, and 57 websites were impacted by it in addition to other WCD vectors, bringing the total to an astounding 456.

We found similar cases, involving three vendors providing customer community management, social media integration, and discussion board services. These were less prevalent in our findings, each impacting less than 10 websites. Nonetheless, this illustrates that WCD exhibiting itself as a supply chain vulnerability is not an isolated incident. As evidenced by the alarming numbers, the security community would benefit from investigating supply chain attacks in a broader scope in the face of novel web cache attacks.

5.1.5 Method Interchange and WCD

Method Interchange refers to the server-side behaviour where a web application processes both `GET` and `POST` requests interchangeably for the same action, typically by treating query string parameters and `POST` body parameters as equivalent. While this may be intended to increase flexibility, it introduces security risks. Exploiting certain vulnerabilities, such as XSS or Cross-Site Request Forgery (CSRF), is often significantly easier via `GET`, since it does not require inducing the victim into visiting a third-party site for exploitation. Moreover, some security mechanisms, such as CSRF tokens or input validation routines, are implemented selectively, often enforced for `POST` requests but overlooked for `GET`. This discrepancy enables attackers to bypass protections by leveraging method interchange, undermining the intended security measures [126].

RFC 7231 defines the HTTP methods and status codes that may be cached [46]. HTTP `GET` and `HEAD` methods can be cached if they have the necessary characteristics, while `POST` method responses can only be cached if they include explicit freshness in-

formation. However, most web caching implementations do not allow caching of `POST` responses, as they are typically not idempotent and may contain sensitive data.

Exploiting Method Interchangeability, we can trick a web cache into caching responses to `POST` requests. This can be achieved by interchanging a `POST` request with a `GET` request that includes the same parameters and payload. This involves moving parameters from the `POST` body to the query string of a `GET` request, or vice versa, and changing the verb of the request accordingly.

If the response is not cached already, we can exploit a WCD vulnerability to forcibly cache the response to the `GET` request. This can be done by applying a WCD payload to an interchanged `GET` request, which will cause the cache to store the response with the sensitive data. The attacker can then retrieve the cached response by sending the same `GET` request, which will return the cached response with the sensitive data.

We demonstrated this attack on a real website, where `POST` requests to a `/graphql` endpoint could be interchanged with `GET` requests. GraphQL is a query language for APIs that allows clients to request specific data from the server. The website in question was also vulnerable to WCD, and we were able to cache the response to a `POST` request by sending a `GET` request with the same parameters and a WCD payload. We reported the issue to the website owner, who promptly fixed it and issued us a €500 bounty for our responsible disclosure.

5.1.6 Section Summary

In this section we have discussed case studies that illustrate the security impact of WCD vulnerabilities. We have shown that WCD can lead to a variety of attacks that go beyond the immediate data leaks, such as cache poisoning, supply chain attacks, and attacks facilitated by method interchange. We showed that WCD can be exploited in various contexts, affecting both unauthenticated and authenticated users, and that it can be used to bypass security mechanisms and cause significant damage. Chaining WCD with other vulnerabilities can lead to sophisticated attacks that are not immediately obvious but can have severe consequences.

5.2 Caching Security Tokens

Content Security Policy (CSP) is a web security mechanism that enables web developers to specify the sources from which their web pages can load resources, such as scripts, style sheets, images, and fonts. CSP is an effective countermeasure to prevent the

exploitation of Cross-Site Scripting (XSS) vulnerabilities, which are one of the most common vulnerabilities on the web. CSP is not designed to be the sole mechanism for preventing XSS attacks, but it is intended as the last layer of a defence-in-depth approach, providing an additional layer of protection for this type of attack without replacing other mechanisms for preventing XSS vulnerabilities, such as sanitization and validation of untrusted input and the use of templating engines. In practice, while other mechanisms aim at preventing the existence of XSS vulnerabilities, the goal of CSP is to render their exploitation impossible, without eliminating the XSS vulnerabilities. CSP2 is a W3C specification, published as a Recommendation by the Web Application Security Working Group [133]. The W3C is currently specifying CSP3 in a Working Draft [137].

By default, the CSP blocks the execution of all inline scripts and styles (i.e., scripts and styles directly included in the HTML code of a web page). However, CSP2 introduced the concepts of *nonces* and *hashes*, enabling websites to allow the execution of specific individual inline scripts and styles without relying on a whitelist. CSP nonces are a random string included in the policy which is assigned to scripts allowed to execute in the form of an attribute (e.g., `<script nonce="r4nd0m">`). Servers are required to generate a new and unique nonce for each response that includes a policy, and the nonces should be generated using a cryptographically secure random generator and should be at least 128 bits long.

In this section, we measure the usage of CSP nonces on popular websites, focusing on nonce reuse. We analyse the causes of nonce reuse to identify whether they are introduced by the server-side code of the website or if the nonces are being cached by a web cache.

5.2.1 Background

We begin by introducing the concept of CSP and nonces, discussing their purpose, how they work, and their role in preventing XSS attacks.

Cross-Site Scripting

Cross-Site Scripting (XSS) is a type of security vulnerability that enables attackers to inject HTML and JavaScript code into the web pages of a vulnerable website. This allows attackers to execute code in the victim's browsers in the context of the vulnerable pages. Typical injection sources are the query parameters or the path of a request URL. By exploiting XSS vulnerabilities, the attackers can steal victims' information, hijack

their sessions and even execute actions on their behalf on the target website. There are three main categories of XSS vulnerabilities based on the injection source.

Reflected XSS happens when parts of an HTTP request are included in the response from the server without sanitization or validation. This type of XSS requires active interaction from a victim. An attacker has to craft a malicious URL that includes a payload and, using social engineering techniques, trick a victim into visiting it.

Stored XSS, instead, arises when user-controlled content stored in a database is included by the web server in the responses to genuine requests by the visitors of a website without being validated or sanitized. To exploit these kinds of vulnerabilities, an attacker has to find a way to save the malicious payload in the server's database, and then wait for the victims to autonomously visit the web pages that include it. Consequently, stored XSS vulnerabilities have generally a higher severity because they do not require user interaction.

The third type of XSS vulnerability is called *DOM XSS* and happens when the user-controlled content is dynamically executed by a client-side script (e.g., JavaScript code). Differently from the other two categories of XSS, the malicious payload is not included in the web page by the server, but it is used to modify the Document Object Model (*DOM*), without proper sanitization or validation [95]. This characteristic of DOM XSS means that it can be performed without the payload having to pass through the server, and can therefore also be exploited in pages coming from a cache.

To prevent XSS vulnerabilities, a common approach is to implement input validation and sanitization to ensure that any user-provided data does not contain malicious scripts and is properly encoded. Specifically, sanitization removes or HTML-encodes potentially dangerous HTML code from users input to prevent XSS attacks, ensuring that the unsafe content is treated as data and not as code [94].

Content Security Policy

XSS vulnerabilities are possible because a web browser has no way of determining whether the injected code included in the web pages is reliable or malicious. The Content Security Policy (CSP) is a security mechanism that helps to prevent XSS attacks, clickjacking, and other code injection attacks by enabling websites to specify what resources can be loaded by the user agent. This mechanism is intended as a defence-in-depth that provides an additional layer of protection against XSS vulnerabilities, by making them not exploitable. CSP can also be used to detect and report code injection attempts, allowing website administrators to take action to prevent further attacks.

The CSP can be deployed in *enforcement mode*, meaning that all the violations of the policy will be blocked (and possibly reported) by the browser, or in *report-only mode*. When a policy is report-only, it is not enforced by browsers, which will instead only report CSP violations to a *report-URI* specified in the policy. The second version of CSP is specified by the W3C in [133], and they are currently working on its third version, CSP3, currently published as a Working Draft [137].

Websites can specify their policy using the `Content-Security-Policy` response HTTP header, or through the `meta` HTML tag. To deploy a report-only CSP, servers must use the `Content-Security-Policy-Report-Only` response header, and cannot do it using the meta tag. The CSP allows website administrators to specify an allowlist of trusted sources for executable content, such as JavaScript, CSS, and images. Any content that comes from untrusted sources must be blocked by the user agent [133].

A CSP is composed of one or more directives separated by a semicolon, and each directive is used to specify the valid sources for a particular type of resource. These directives can be used to allowlist specific domains or sources from which resources can be loaded, and to block the ones that do not match the specified sources. The `default-src` directive is used as a fallback, specifying the default behaviour for any directive that is not explicitly specified. By default, the CSP blocks all inline scripts and styles, effectively preventing the exploitation of XSS vulnerabilities.

CSP Nonces

The second version of this security mechanism, CSP2, introduced the concept of *nonces* and *hashes* as a way of allowing the execution of individual inline scripts and styles without relying on a whitelist. A *CSP nonce* is a random value used only once, which is added to the inline scripts or style tags of a web page as an attribute. The nonce is included in the policy, which tells the browser to only execute scripts or styles that have a matching nonce value [133]. For example, the following policy:

```
Content-Security-Policy: default-src 'self';
    script-src 'nonce-cmFuZG9t' 'self';
```

allows the execution of the first inline script in the following example, but blocks the second and the third scripts because the nonce is missing or invalid, respectively.

```
<script nonce="cmFuZG9t">
    console.log("This will execute");
</script>
```

```
<script>
    console.log("This will *not* execute");
</script>

<script nonce="attacker">
    console.log("This will also *not* execute");
</script>
```

The CSP2 specification [133] indicates that a nonce should have the following characteristics:

1 Must be unique for each HTTP response that includes a CSP.

2 Should be generated using a cryptographically secure random number generator.

3 Should be at least 128 bits long, before encoding.

A CSP nonce should be long and randomly generated to prevent attackers from guessing or brute-forcing it. However, it must be noted that properly implementing a nonce policy does not necessarily prevent the exploitation of XSS vulnerabilities altogether. In fact, if an inline script with the correct nonce attribute uses untrusted user input, an attacker could bypass the policy and achieve XSS.

If a CSP nonce is reused for multiple HTTP responses, an attacker that is able to steal it could effectively bypass the policy by injecting a script with the correct nonce value.

5.2.2 Related Works

The Content Security Policy was proposed by Stamm et al. in 2010, as an additional layer of security for XSS and CSRF attacks [119]. Numerous studies have measured the adoption of CSP by websites in the wild and its variation over time, giving us an indication of how much its use has grown through the years. In 2014, Weissbacher et al. measure the adoption of CSP in the Alexa Top 1M over a period of 16 months and find that only 850 of the use it, 815 of which are in enforcement mode [136]. Calzavara et al. do the same in 2016 [15] and 2018 [17], finding respectively 8,133 and 16,353 sites using the CSP, marking a significant increase in its adoption. Finally, Roth et al. present a long-term study on CSP adoption from 2012 to 2018 by inspecting the headers of 10,000 websites using the Internet Archive, finding that 1,233 of them deployed a CSP for at least one day in the analysis period [103]. They also measure for the first time the adoption of CSP nonces and hashes, newly introduced in the second version of CSP's

specification [133], finding that in 2018 only 5% of sites adopting a CSP used nonces and 1% hashes. They also show that 13% of sites whitelisted domains that are expired, contain typos, or resolve to private IP addresses. Finally, they argue that CSP adoption is lagging behind because it has a bad reputation for being complex, and document the developers' struggles while trying to deploy it properly.

In 2013, Doupé et al. present deDacota, an automatic tool to statically rewrite code to separate data from the code and enforce this separation at run-time using CSP [35]. The following year, Johns analyses the danger of dynamically generated scripts on the server side to fill in values in scripts with data retrieved at run-time. He proposes PreparedJS, a templating mechanism that allows separating dynamic data values from script code, and a script checksumming scheme, enabling the server to communicate which scripts are allowed to run to the browser [68]. In the same year, Weissbacher et al. compare the adoption of CSP with other security headers, investigate the challenges of adopting a CSP, and experiment with a semi-automated crawler-based CSP generation mechanism [136].

Fazzini et al. further explore the automated generation of Content Security Policies for existing web applications and present *AutoCSP*, an automatic tool that uses dynamic taint analysis to identify the trusted content of dynamically generated HTML pages that should be allowed to be loaded, rewrites the server-side code that generates those pages and develops a policy accordingly [40]. In 2016, Kailas and Braun analyse the usage of CSP in real-world websites and identify errors and inconsistencies. They also present a tool to automatically generate valid policies on the client side, targeting savvy users [98]. In the same year, Pan et al. propose *CSPAutoGen*, a tool to facilitate the adoption of CSP by websites that auto-generates policies in real-time, without requiring modifications to the server-side code. Their tool generates policies according to the page content and the templates of scripts, which are based on the Abstract Syntax Tree (AST) representing the scripts and a type system. Their idea of templating is based on the fact that dynamically generated scripts generally share the same structure and differ only in their data values [96]. Kerschbaumer et al. show that 90% of CSP deployments include the `unsafe-inline` keyword, rendering it ineffective in preventing XSS attacks, and present a system to automatically generate CSPs by whitelisting only the expected scripts on a site using hashes, identified using a crowd-sourced approach [70].

In 2015, Hausknecht et al. analyse the interplay between the CSP and browser extensions, finding that some extensions tamper with the CSP of websites to be able to work. In 2017, Some et al. analyse how the SOP could cause CSP violations [117], and in 2021 Steffens et al. analyse how the use of third-parties resources impacts a website's

implementation of CSP and argue that relying on third parties is a major roadblock for security [120]. In 2021, Roth et al. present a qualitative study that analyses the difficulty of developing a safe CSP by tasking real-world developers with an interview that includes a drawing task, where participants have to draw and explain an XSS attack, and a programming task, where they have to develop a CSP for a small web app to prevent XSS attacks [104].

Calzavara et al. in 2016 analyse four aspects that affect the effectiveness of CSP, finding that CSP has limited deployment and that the deployed policies exhibit several weaknesses and misconfiguration errors. They also measure the most used features of CSP and how many sites only use it in report-only mode [15]. In 2017, Calzavara et al. propose an extension to CSP called *Compositional CSP* (CCSP) that enables the composition of policies at run-time and assess its potential impact in the wild. They implement a Proof-of-Context of CCSP as a Chromium extension and write CCSP policies for a few popular sites which normally trigger CSP violations [16].

In 2016, Van Acker et al. show how to exploit DNS prefetching to exfiltrate data, bypassing the protection offered by the CSP [127].

Weichselbaum et al. present a thorough analysis of the practical benefits of CSP, mainly related to the prevention of XSS vulnerabilities exploitation, and analyse real-world policy deployments to identify common flaws. They find that in 2016 only 0.16% of the domains they analysed use CSP and detect that 94.7% of distinct policies deployed can be bypassed. They also find that 90.6% of policies contain configurations that remove any protection from XSS attacks, and that 51% of the remaining 9.4% can still be bypassed due to policy misconfigurations or origins with unsafe endpoints in the `script-scr` whitelist. They argue that maintaining a secure whitelist for a complex application is infeasible in practice, and suggest replacing URL whitelisting with nonces and hashes [135]. Moreover, they propose a new CSP source expression called `strict-dynamic`, enabling dynamically generated scripts to implicitly inherit the nonce from the trusted scripts that generated them, easing the adoption process of nonces-based policies, which is now included in the W3C CSP3 Working Draft [137]. In 2017, Lekies et al. present a novel attack that exploits *script gadgets*, i.e., small fragments of JavaScript in a site's legitimate code that can be used to bypass XSS protection mechanisms, including the CSP. They find that gadgets are extremely prevalent on the web, setting a lower-bound at 19.9% of sites [80].

5.2.3 Methodology

We now describe the methodology we developed to detect websites that deploy a Content Security Policy and reuse the same CSP nonce in different responses. We check if the nonce reuses are caused by a web cache or if they are due to the server-side code.

URLs Collection

First, we collect a list of URLs to test on a website. We crawl a website in an unauthenticated way, starting from its homepage and recursively following all the links to other pages, excluding any links that lead to external domains. Our crawler is configured to only visit a limited number of pages on a limited number of domains and subdomains to avoid overloading the target website.

CSP Detection

To identify the URLs on a website that present a CSP, we look for both the response header `Content-Security-Policy` and the `meta` tag in the HTML of the page. We also test whether the CSP is deployed in report-only mode by checking the presence of the `Content-Security-Policy-Report-Only` header.

Nonce Detection

To detect the use of nonces in the CSP, we check if the page includes any `script` tags with a `nonce` attribute. We save the HTML source code of the pages that include a CSP nonce, and store the request and response headers for future analyses. Since we are interested in sites where nonces are actually used, we do not only check how many sites include a nonce in their policy, but also make sure that nonces are used in script tags.

Nonce Reuse Detection

Next, we analyse all the previously identified CSP nonces to check if they are reused in multiple responses. To detect nonce reuse, we request the same URL twice and check if the value of the nonce is the same. If it is the same in both responses, we mark the page as reusing nonces and investigate its cause.

Nonce Reuse Cause Detection

To determine the cause of nonce reuse, we employ three mechanisms that, used together, give us a strong indication of whether a cache is responsible for the nonce reuse or not.

Static Nonces First, we attribute to the server-side code all the nonce reuses where the nonce is the same in all responses coming from a website, even on different paths. We perform this test only on the sites that use a nonce on two or more pages. While these nonces may still get cached, the root cause of the reuse is the server-side code, which is generating the same nonce for all responses.

Cache Busting Next, we perform a third request including a cache-busting query parameter in the URL. This check allows us to detect nonce reuse not caused by a web cache. This technique is effective when the query string is keyed. In these cases, adding a random modification to the query string forces our request to have a different cache key, effectively preventing the cache from serving us with a cached response and a cached nonce value. If the response to the cache-busted request includes a different nonce, we attribute the nonce reuse to a web cache. However, if the value of the nonce is the same as the one in the previous responses, we cannot conclusively exclude the influence of a cache, as some web caches might not include the query string in their cache key. In this case, we need to perform a third check.

Cache Header Heuristics In this test, we check whether the response with a reused nonce is coming from a cache or the origin server using the *Cache Headers Heuristics* that we described in Section 3.2. If the response to our second request is coming from the origin server, we conclusively attribute the nonce reuse to the server-side code. Otherwise, if the second response is coming from a cache or if the test gives an unknown status (e.g., because the cache status headers are missing in the response), we cannot conclusively attribute the nonce reuse to a cache or not, and we need to resort on cache busting. If we detect that the second response is coming from a cache, we cannot exclude that the server-side code is nevertheless issuing the same nonce.

In summary, only by using all three tests in combination we can attribute nonce reuse to a cache or not with a high degree of confidence.

Table 5.1: The number of sites that deploy a Content Security Policy, use CSP nonces, and reuse the same nonce value for multiple responses. Percentages are calculated over the total number of sites that deploy a CSP (10,034).

Total sites using CSP	**10,034**	
enforcement mode	8,946	(89.2%)
report-only mode	1,088	(10.8%)
Sites with CSP nonces	2,271	(22.6%)
Sites reusing CSP nonces	598	(6.0%)

5.2.4 Experiment

Using the methodology described above, we perform a large-scale experiment on the web to detect CSP nonces reuse. We use the Tranco Top 50k [99] generated on July 21, 2022. [1] For each domain in our dataset, we crawl at most 10 pages on at most 10 FQDNs. This results in a maximum of 100 pages for each domain in our dataset. For each domain that we test, we create a session object with cookies persistence to simulate a user surfing a website using the same browser for all requests.

5.2.5 Results

Table 5.1 summarizes the results of our measurement of CSP adoption.

While it is not the main focus of our research, we briefly analyse the adoption of CSP. We crawled 50k websites and, of those, we detected 10,034 (20.1%) sites deploying a Content Security Policy. Of these, 1,088 sites only use the CSP in report-only mode, while 8,946 enforce their policy. We also found 346 sites only deploy their policy in the `meta` tag, while 135 deploy it both in the *Content-Security-Policy* header and the `meta` tag.

Nonce Reuse

Table 5.2 presents the results of our measurement of websites that use a CSP and that reuse nonces. Of the 10,034 sites that deploy a CSP, 2,271 (22.6%) present use nonces in at least one of the pages that we visited. 598 (26.3% of the sites using a nonce-based CSP) reused the same CSP nonce value in multiple responses. Figure 5.1 shows the distribution of websites that use a nonce CSP in at least one of their pages and the sites that reuse at least one nonce, with respect to their ranking in the Tranco Top 50k. We

[1] Available at https://tranco-list.eu/list/W97W9/

Table 5.2: The number of sites presenting at least one reused CSP nonce and the investigated reason of the reuse. Percentages are calculated over the total number of sites that reuse a nonce (*598*).

Total sites reusing nonces	**598**	
due to a cache	256	(42.8%)
server-side code	342	(57.2%)
in the same session	37	(6.2%)
in different sessions	561	(93.8%)

can see that the usage of CSP nonces is higher among more popular websites, while the percentage of websites that reuse the same nonce in more than one response is higher among the less popular ones.

We then find that 342 (57.2% of the 598 sites reusing a nonce) sites use the same nonce in more than one origin server response, indicating that the server-side code is responsible for this behaviour, while for 256 (42.8% of the 598 sites reusing a nonce) sites the cause of nonce reuse is solely due to a cache. In these cases, it is the cache storing and serving a copy of the response that includes an otherwise dynamic nonce. 219 sites have static nonces, meaning that they use the same nonce value in all responses coming from the origin server.

Using cache-busting and the cache header heuristics, we also detect 318 websites that cache CSP nonces (53.2% of the sites that reuse nonces), regardless of the nonces being static or dynamic when generated by the server-side code. Of these 318 cases where we observed caching, 190 were detected using cache-busting, and 128 using the CHH algorithm.

5.2.6 Ethical Considerations

During this research, we took several precautions to ensure that our methodology and testing approach was conducted responsibly. Firstly, we ensured that our testing did not cause harm to any real users of the websites we tested. We conducted our testing solely on publicly accessible pages and did not inject any malicious payload. In fact, during this research, we did not exploit any XSS vulnerability, as it is out of the scope of this project. Secondly, we limited the number of tests per website to a maximum of 100 pages and performed a maximum of 4 HTTP requests per page. This was to prevent excessive website load and avoid any potential negative impact on website performance or availability. Furthermore, we timed our requests not to send multiple requests one after the other.

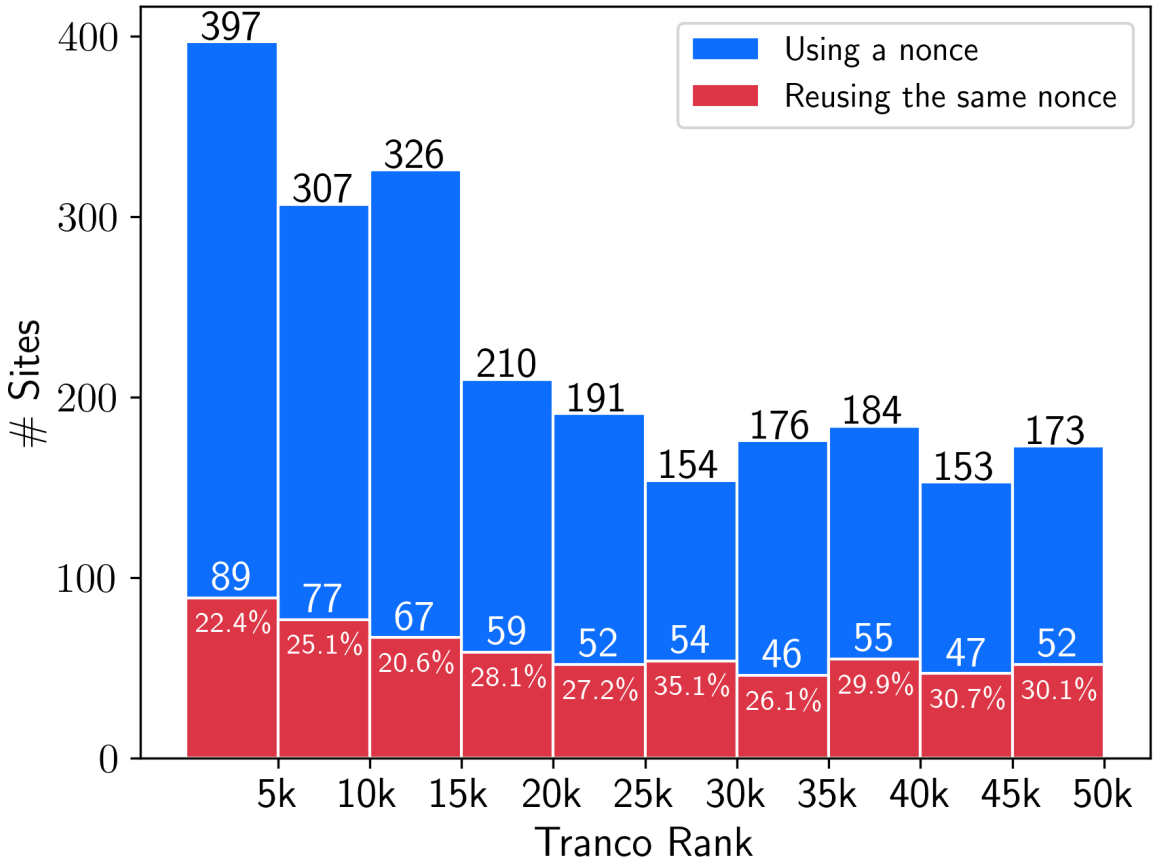

Figure 5.1: The distribution of websites that have a nonce-based CSP (in blue), and the subset of those which reuse a CSP nonce (in red) with respect to their ranking in the Tranco Top 50k. The percentage in each bar is calculated over the number of sites that use a nonce in the same 5k bucket.

5.2.7 Mitigations

Reusing the same nonce value is a dangerous behaviour by websites that can effectively hinder the protection offered by the CSP from the exploitation of XSS vulnerabilities. Reusing the same nonce is in some cases the same as allowing all scripts inline, in others, it is a severe relaxation of policy with a dramatic reduction in the protection offered. Implementing a proper nonce-based policy is a complex and costly task, but it is the only way a website using it can fully protect itself against XSS.

To mitigate the risk of nonce reuse, we recommend that website developers follow the guidelines provided by the CSP specification [133] and ensure that they generate a new and unique nonce for each response that includes a CSP. This can be achieved by using a cryptographically secure random number generator to generate the nonce value, and by ensuring that the nonce is at least 128 bits long before encoding it in base64.

Avoiding nonces from being cached is also important to prevent DOM XSS vulnerabilities from being exploited. To achieve this, website operators should make sure that responses containing a nonce are not cached by web caches. This can be done by setting the `Cache-Control` header to `no-store` or `no-cache`, or by using the `Vary` header to indicate that the response should not be cached based on the nonce value. The cache should also be configured to honour the `Cache-Control` header, which is a common, and sometimes default, configuration for many web caches [87].

5.2.8 Section Summary

In this section, we presented a measurement of several aspects of the adoption of the Content Security Policy by popular websites. We found that, compared to previous large-scale measurements, the adoption of CSP has grown. At the same time, the adoption of nonces-based policies is also growing. However, our research highlights that more than one in every four sites that make use of nonces, uses them incorrectly, using the same value in more than one response. Reusing the same nonce more than once could result in a complete bypass of the CSP, allowing attackers to exploit possible XSS vulnerabilities. We analysed the different conditions in which nonces reuse happens and investigated its possible causes. Depending on the cause, it may be easier or harder for an attacker to bypass CSP to exploit XSS vulnerabilities.

Specifically, if the same nonce value is reused for different users of a website, it is effectively the same as deploying an `unsafe-inline` directive, allowing the execution of all inline scripts and styles, regardless of their origin; an attacker can simply visit the vulnerable website to obtain a valid CSP nonce value to craft an injection that bypasses the CSP. On the other hand, if a nonce is only reused for a single session, the attacker has to exploit other vulnerabilities or attacks to steal the nonce value from the target victim. For example, an attacker can exploit possible Web Cache Deception vulnerabilities or perform cross-origin requests (if the *SameSite* attribute of the cookies is set to *None*). Once the attacker holds a nonce linked to the session of a victim, they can craft an injection that includes the stolen nonce to exploit an XSS vulnerability.

Next, we analysed the causes of nonce reuses to detect the ones likely caused by the presence of a web cache storing otherwise dynamic nonces. If a nonce is reused due to a cache, it is harder for an attacker to bypass the CSP and achieve XSS. In fact, even if an attacker can steal a cached nonce of a victim, they cannot inject a malicious payload in the cached copy of the response directly. The only way an attacker can achieve XSS under these conditions is if, in addition to stealing a victim's cached nonce, they

find a way to inject a payload with the stolen nonce by exploiting a client-side XSS vulnerability. This happens when untrusted data is dynamically loaded directly from the victim's browser when the page is vulnerable to DOM XSS, or when the page includes data dynamically (e.g., through XMLHttpRequest or AJAX).

5.3 Chapter Summary

In this chapter, we have explored the uncommon consequences of Web Cache Deception vulnerabilities. We have shown that WCD can lead to a variety of attacks that go beyond the immediate data leaks, such as cache poisoning, supply chain attacks, and attacks facilitated by method interchange. These attacks can have severe consequences, affecting both unauthenticated and authenticated users, and can be used to bypass security mechanisms and cause significant damage.

Next, we have presented a measurement of the adoption of the Content Security Policy by popular websites, focusing on the use and reuse of CSP nonces. We found that, while the adoption of CSP has grown, more than one in every four sites that make use of nonces, uses them incorrectly, using the same value in more than one response. Reusing the same nonce more than once could result in a complete bypass of the CSP, allowing attackers to exploit possible XSS vulnerabilities. We analysed the different conditions in which nonces reuse happens and investigated its possible causes, attributing them to the server-side code or to a misconfigured web cache. Depending on the cause, it may be easier or harder for an attacker to bypass CSP to exploit XSS vulnerabilities.

This chapter highlights the importance of deep understanding of the working of web caches and their interaction with web security mechanisms to minimize the risks posed by adopting web caches and to mitigate the risks of WCD vulnerabilities. It also shows that WCD can have uncommon consequences that are not immediately obvious, but can be severe, and that it is important to consider them when assessing the impact of WCD vulnerabilities and when designing mitigations for them.

Chapter 6

Web Cache Overflow: Exploiting Imprecise Cache Keys for DoS and Beyond

This chapter is based on works currently under review at Euro S&P 2026 as:

Golinelli, M., Onarlioglu, K., & Crispo, B. (2025). Web Cache Overflow: Exploiting Imprecise Cache Keys for Denial of Service and Beyond.

As previously discussed, web caches have reached existential importance in meeting the low access latency requirements for clients, and traffic offload demands for origin servers. Cache degradations can rapidly escalate to costly network congestions and Denial of Service (DoS) incidents, and their efficient utilization is crucial. The topic of caching has been a staple of computer science, with researchers exploring different caching strategies, cache replacement algorithms, and even adversarial techniques to impair cache efficacy or launch cache-based side-channel attacks for various hardware and software engineering contexts. However, web caches remain particularly susceptible to abuse due to two important factors: Their immense exposure to *untrusted input* that influences the caching decisions, and widespread web cache configuration antipatterns that do not account for such potential abuse.

Cache keys are often not *precisely* defined. For example, a website operator who creates a caching rule for a static image object served from `"example.com/pic.jpg"` may simply define the cache key as the full URL, overlooking the possibility of query strings

included in the URL. This oversight would result in a subsequent request for an equivalent but syntactically different URL, such as `"example.com/pic.jpg?junk=123"`, yielding a different cache key, because the two URLs do not exactly match. If discovered, a client can then take advantage of this situation to craft a multitude of requests for the same object, but with different cache keys, causing the web cache to retrieve the object from the origin anew every time. This is an instance of a well-known trick called *cache busting* that we extensively discussed in previous chapters, often used by web developers for testing application changes, without needing to purge caches to observe the effects of their changes.

Motivated by these observations that clients have complete influence over the HTTP requests used to derive cache keys, and that imprecise cache key definitions are commonplace, we hypothesize that attackers can systemically and systematically weaponize seemingly trivial cache busting tricks, and launch devastating pollution attacks on web caches. Specifically, attackers can rapidly fill caches with large amounts of duplicate objects, forcing the eviction of everything else, and therefore negating the benefits of using a web cache. In this chapter, we present *Web Cache Overflow (WCO)*, a novel cache pollution attack that exploits imprecise cache keys to fill a web cache, resulting in legitimate cached objects being purged. WCO is built on the cache busting concept that we discussed earlier in Chapter 3.

One obvious impact of WCO is service degradation, which rapidly escalates to a DoS. Moreover, since web caches are frequently deployed in shared cloud infrastructures and hosting providers, an attack resulting from one tenant's imprecise cache keys may DoS all tenants of the platform. However, attackers can also leverage the capability to purge popular cached objects to other nefarious ends. For example, *cache poisoning* attacks that aim to trick caches into storing malicious content are on the rise. Such attacks are challenging to launch against frequently accessed objects (e.g., a JavaScript file embedded on the home page), as such objects are often perpetually cached due to their popularity. With WCO, an attacker can fill the cache to trigger purging of even popular objects, exposing a window of opportunity to poison that object's now unused cache key with malicious content.

One novel (and concerning) characteristic of WCO is that, unlike the rest of the cache attack literature, it requires no prior knowledge of the caching algorithm's intricacies or a calculated abuse of cache locality. Instead, it is an attack enabled by the fundamental workings of HTTP and web caches, it can be exploited by any unauthenticated, unauthorized, unsophisticated Internet client, and it is difficult to mitigate without incurring significant costs to website operators.

We test these claims through a detailed evaluation of WCO. We first conduct experiments with a selection of popular caching proxies to show that the attack is implementation agnostic, and then perform focused tests using NGINX , exploring the influence of parameters such as the cache store capacity and cached object size. Testing WCO against production websites is not ethically feasible, as DoS attacks are disallowed in bug bounty programs. In lieu of penetration testing, we conduct a measurement study on the Tranco top 10K domain list to identify the presence of exploitable objects cached under imprecise cache keys. Detailing the cases where the attack succeeds and fails, we demonstrate that WCO impacts many practical cache deployment setups.

We finally provide a discussion of seemingly intuitive mitigations, namely, cache store quotas, request rate limiting, and object deduplication, but offer evidence through additional experiments that these are either ineffective, counter-productive, or unreasonably costly. We conclude that addressing the issue's biggest contributing factor by defining *precise* cache keys is the most promising defense, and we document this as a security best-practice for the first time.

Code and Data Availability

We release two open-source tools: a penetration testing utility for detecting imprecise cache keys, intended for website operators and security testers, and an implementation of our attack to evaluate systems against WCO. All the code and data required to reproduce our experiments are also available in our anonymous repository at `https://anonymous.4open.science/r/wco-154E`.

Chapter Outline The chapter is structured as follows. We start by discussing related works and providing background on cache pollution attacks in Section 6.1. In Section 6.2, we present the problem of imprecise cache keys that enables WCO, and discuss its impact and novelty compared to prior works. We then describe the methodology for launching WCO in Section 6.3, followed by a detailed evaluation of the attack in Section 6.4. Finally, we conclude with a discussion on potential mitigations and best practices.

6.1 Background and Related Works

We start by discussing cache pollution and presenting works related to WCO. Readers can refer to Chapter 2 for a general background on web caches and cache busting.

6.1.1 Cache Pollution

Achieving good cached data locality through optimizing access patterns, adopting the appropriate cache eviction strategies, and designing cache hierarchies have been studied in computer science for decades. There is an immense body of literature on this topic as it applies to software, hardware, and network engineering domains. These works define the term *cache pollution* to broadly refer to conditions that lead to a disruption of the cached data locality, resulting in a decrease in cache hits, and therefore performance degradation. Cache pollution is not necessarily an adversarial event, but rather, an outcome of less-than-optimal cache design given the use case. As an early but representative example relevant for a web application context, researchers showed that automated access patterns of web robots significantly decreased cache locality for popular objects [5].

Cache pollution can also be induced for malicious purposes, aptly called a *cache pollution attack*. Core attack concepts generally apply to all caches, and the adversarial techniques presented in all works can be summarized in two categories: 1) Attacks that repeatedly request the same unpopular objects, and 2) attacks that request a wide array of objects. Both techniques hurt the locality of genuinely popular objects. Defences fall into various anomaly detection approaches based on access characteristics and patterns, and specialized cache replacement algorithms to minimize attacker influence.

For network applications of caches, the vast majority of literature is focused on Information-Centric Networking (ICN), which encompasses Content-Centric Networking (CCN) and Named Data Networking (NDN). ICN is a novel network communications paradigm for the Internet that revolves around requesting named content, instead of host-to-host packet exchanges [139]. In-network caching plays a prominent role in this architecture, which resulted in a plethora of scholarly works exploring pollution attacks and defences for the ICN paradigm (e.g., [79, 141, 85, 28, 144, 58, 143, 97, 59, 69]). We omit the specifics of individual works, as ICN is not relevant to this thesis' scope. However, the attack and defence contributions are in line with the general summary we provided above.

Gao et al. and Deng et al. present incremental works more closely related to ours, exploring cache pollution on the web [48, 30]. These discuss the same two attack categories as applied to caching proxies and DNS caches, propose an anomaly detection scheme based on statistics computed over request features, and perform experiments using Squid. They primarily focus on pollution attacks in web and peer-to-peer cache deployments using forward proxies, as opposed to the *reverse* proxies more prevalent

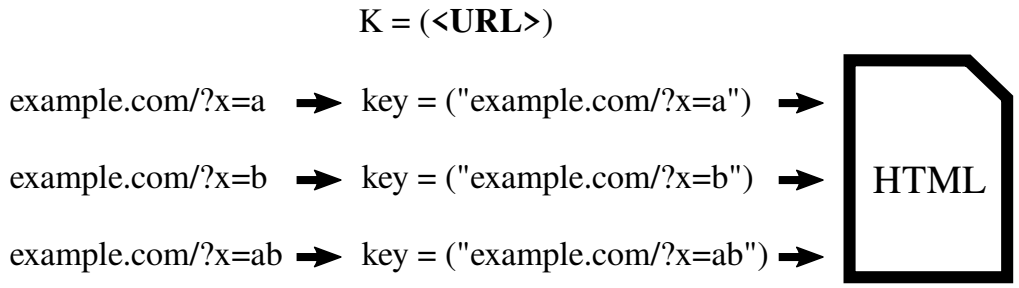

Figure 6.1: Cache pollution due to imprecise cache key definition, keyed on the full URL.

today that we focus on. Nonetheless, the authors argue (and we agree) that the findings are also applicable to reverse proxies. More recently, Afek et al. explore a similar attack against DNS caches [2]. While the idea of flushing a cache via adversarial input is shared, the operational characteristics, defences, and implications differ significantly between DNS and web caches.

6.2 The Problem of Imprecise Cache Keys

The observation that motivates our research is that, while cache busting is often seen as a harmless–and sometimes even useful–quirk of cache-enabled web architectures, the factors that enable cache busting can trivially be weaponized by an attacker to launch devastating cache pollution attacks.

We elaborate on the details of this observation and our resulting hypothesis in more detail below.

Let:

- $K = (k_1, ..., k_n)$ be the n-tuple that represents the cache key definition, where $k_i \in K, i = 1, ..., n$ describe the keyed HTTP request elements.
- $request_i$, $i \in \mathbb{N}$ be *distinct* HTTP requests that result in cacheable responses containing payloads $object_i$.
- key_i be the concrete cache keys derived from $request_i$ according to the definition K, resulting in the cache store mapping $key_i \rightarrow object_i$.

Then:

For any two $key_i \rightarrow object_i$ and $key_j \rightarrow object_j$,

if $key_i \neq key_j$, but $object_i = object_j$,

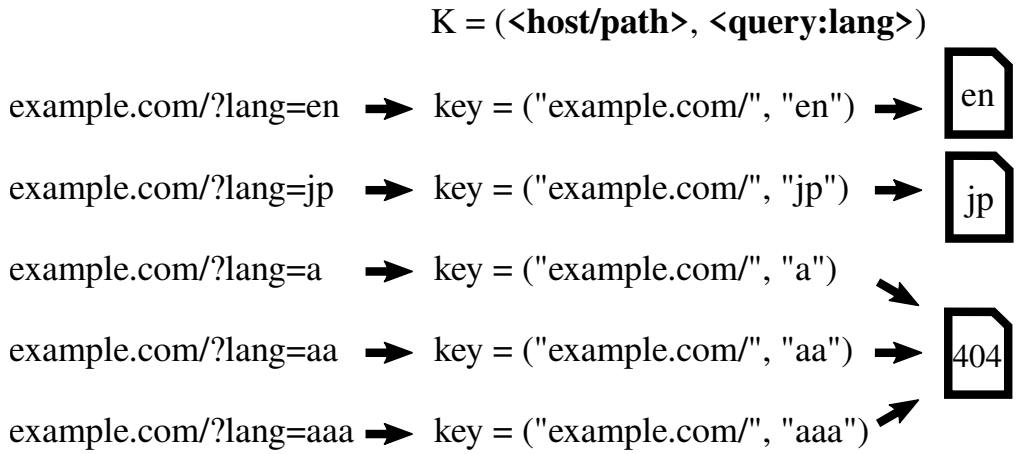

Figure 6.2: Cache pollution due to imprecise cache key definition, keyed on the query string parameter "lang", combined with insufficient validation of its value.

we say that K is *imprecise*.

Intuitively, a cache key definition that includes keyed request elements that do not influence the response payload results in the identical object getting stored multiple times, but under different cache keys, wasting space. This is a generalization and formalization for the basis of all cache busting, which may then be systematically abused for cache pollution. Web clients are free to craft and send any HTTP request to a server, and therefore, they can manipulate any keyed element included in K in an attempt to achieve this effect.

The semantics of a *precise* cache key definition naturally follow: Every unique concrete cache key derived from a precise definition maps to a unique cached object, eliminating this cache pollution vector.

Note that an imprecise cache key definition is a significant contributing factor, but it is neither a root cause nor a sufficient condition for abuse. Whether cache pollution via imprecise cache key definitions is viable depends on other factors such as what validation a server performs on requests, and what responses are configured to be cacheable either through cache rules or `Cache-Control` headers.

Figure 6.1 depicts an example where the imprecise cache key definition leaves little room for mitigating abuse through validation. Here, the cache key is defined as the entire URL, including the host, path, and the complete query string. As a result, an attacker can attach arbitrary query strings that are not recognized by the application to repeatedly cache the homepage under different cache keys, indefinitely.

Figure 6.2, on the other hand, illustrates a more complex situation. The cache key is now defined more prescriptively to only include the host, path, and a specific query

string parameter that the application uses to serve a homepage translated into different languages. This mitigates the attack in the previous example, since arbitrary query strings do not influence the cache key. However, there is still room for abuse: If 1) the application does not validate the parameter against a list of supported languages, but allows any arbitrary string, and 2) the cache rules allow caching of 404 error pages–a common practice for traffic optimization–, then the attacker can still pollute the cache by repeatedly caching the same error page under different values for "lang".

Given the above observations, we present our hypothesis: An arbitrary Internet attacker can abuse imprecisely defined cache keys, keyed on insufficiently validated HTTP elements, in order to launch cache pollution attacks that fill a web cache to capacity with redundant copies of an object. We call this attack *Web Cache Overflow* (WCO).

6.2.1 Impact And Novelty

WCO fills a web cache with redundant copies of a specific object, forcefully purging everything else in the process. This negates the benefits of using a cache, drastically increasing the traffic load on the origin server and round-trip times for clients interacting with the application, ultimately resulting in a DoS similar to other cache pollution attacks.

However, WCO has a significant property that differentiates it from previous work: it requires no knowledge of the victim cache's regular access patterns, the popularity of files served from the target website, or the cache eviction algorithm in use. Existing attacks rely on measurements and accurate estimations of such details for calculated, complex strategies that selectively boost the locality of unpopular files or disrupt the locality of popular ones, whereas such information is immaterial for WCO. The attack methodology requires no prior knowledge of the target environment and instead results in indiscriminate purging of all objects, making it trivially launched by even unsophisticated attackers. The sole requirement, identifying imprecise cache keys, can be trivially automated, as we discuss in Section 6.3 and later demonstrate in Section 6.4.1. Moreover, our work presents a fundamentally different attack vector and threat model compared to prior studies: previous attacks rely on pre-existing unpopular objects to fill the cache, inherently limiting their scale to the number and size of such objects available on the target site, whereas our approach employs cache-busting techniques to actively create arbitrary cache entries, removing this constraint. Thus, our attack is limited not by the site's content but only by factors such as available bandwidth, server-side rate

limiting, or the expiration age of cached entries. Finally, our methodology avoids the high cost of downloading large files by leveraging HEAD requests, further reducing the overhead of launching the attack.

Furthermore, WCO's ability to purge any and all objects facilitates other attacks such as cache poisoning and cache-based side channels, which usually come with a pre-condition attached: The resource targeted with these attacks must not already be cached. Assume an attacker discovers a reflected cross-site scripting (XSS) vulnerability, where the application includes parts of the attacker-controlled request on the rendered page without input validation or output sanitization, allowing the attacker to inject a malicious JavaScript snippet into the page. If this page is also cacheable, the attacker has an opportunity to cache the page rendered with the reflected XSS payload, escalating the attack to a stored XSS via cache poisoning.

While such vulnerabilities are common, exploiting them is challenging if the page in question is getting a lot of traffic, implying that the original copy will perpetually be cached, not allowing the attacker to override it with a poisoned version. With WCO, however, the attacker is now equipped with a technique to force the eviction of this object, creating a window of opportunity to poison the cache.

6.2.2 Threat Model

We adopt a standard web application threat model. The attacker is any arbitrary Internet user, in full control of their user agent, with the capability to craft any HTTP request, and contact any web application fronted by a public web cache exposed on the Internet. Whether the transport is secure or not is irrelevant for our work. We make no further assumptions.

6.3 Methodology

We now describe how to achieve WCO in concrete steps. Conceptually, WCO is straightforward, but there are important considerations and optimization opportunities hidden in details. We first provide a simplified view into the core methodology for brevity, and later discuss those considerations.

Algorithm 5 Pseudo-code for the high-level WCO flow.

Input *target*: Victim cache-fronted website.

```
1: path, keySpec ← findObjectWithImpreciseKey(target)
2:
3: # Procedure 1: Fill the cache with redundant objects
4: busterList ← []
5: repeat
6:     busterRequest ← generateCacheBuster(path, keySpec)
7:     sendGET(busterRequest)
8:     busterList.append(busterRequest)
9: until isCacheFull() = True
10:
11: # Procedure 2: Refresh the TTL of cached objects
12: loop
13:     for all busterRequest ∈ busterList do
14:         response ← sendHEAD(busterRequest)
15:         if response.cacheStatus = Miss then
16:             sendGET(busterRequest)
17:         end if
18:     end for
19: end loop
```

6.3.1 Overview

WCO follows the below steps, also illustrated in Algorithm 5 with specific lines referenced in the text.

1 Pick a target victim website fronted by a web cache. Prior knowledge of the cache capacity, cache eviction algorithm, or traffic patterns to the website is **not** necessary. However, if this information is available, it can be used to optimize the attack.

2 Identify a cacheable object specified by an imprecise cache key (line 1). This is the object that the attack will fill the cache with, and large files are better.

3 Abuse the imprecise cache key definition to generate a unique cache buster; i.e., a request for the cacheable object with the appropriate keyed element modified to cause a cache miss, resulting in a fresh copy of the object being fetched from the origin and stored under a new cache key. Repeat this process until the cache is approximately filled to capacity with redundant objects. Also keep a list of all generated cache busters (lines 3-9).

At this stage, the origin is suffering the full consequences of the attack due to cache degradation resulting in repeated cache misses for legitimate traffic. The

number of entries in the list of cache busters we maintain matches the number of objects needed to fill the cache.

4 To maintain the attack and prevent legitimate traffic from eventually recovering the cache locality for popular objects, cycle through the list of cache busters used for filling the cache in the previous step, and probe them with HEAD requests (lines 11-19). The HEAD request refreshes the cache time-to-live (TTL) of the bogus object, preventing it from getting evicted.

5 If the response for any HEAD request indicates a cache miss, that implies that the corresponding bogus object was already evicted. Therefore, restore it to the cache by sending a new GET request with the same cache buster (lines 15-16).

In Algorithm 5, the routines *sendGET* and *sendHEAD* are self-explanatory. We discuss the undefined routines *findObjectWithImpreciseKey* and *isCacheFull* later in the following subsections.

6.3.2 Cost Considerations

As with any volumetric attack, WCO is only meaningful when the damage inflicted on the origin server outweighs the cost incurred by the attacker. We define the metric *Damage Ratio (DR)* to measure the effectiveness of the attack based on the disruption of the cache hit rate. DR is calculated by measuring the increased traffic that reaches the origin server due to cache misses resulting from the attack, divided by the attacker's bandwidth expense. A higher value indicates greater disruption relative to the attacker's resource investment.

$$\mathsf{DR} = \frac{T_{\text{attack}} - T_{\text{normal}}}{T_{\text{WCO}}}$$

where:

- T_{normal}: Volume of traffic reaching the origin server due to cache misses under normal conditions.
- T_{attack}: Volume of traffic reaching the origin server due to cache misses during the attack.
- T_{WCO}: Traffic generated by the attacker.

The target cache's capacity and size of the redundantly stored object directly factor into the cost, as they influence the number of requests necessary to fill the cache and maintain the attack. Consequently, the use of HEAD requests during attack maintenance is intentional. Since responses for HEAD requests do not contain the body payload, this greatly reduces the client-side bandwidth, and therefore the attack cost.

Beyond these general considerations, the client-side bandwidth may be further reduced by exploiting the implementation quirks of the victim cache server. We identified two such opportunities in our experiments.

First, servers may be configured to perform opportunistic caching by upgrading a client's HEAD requests to GETs before forwarding them to the origin, and storing the resulting object without transferring it back to the client. If targeting such a cache, all GET requests in our methodology can be replaced with HEADs and lines 15-17 in Algorithm 5 eliminated, resulting in the same disruption, but without making the attacker incur the cost of receiving the response payload. Our experiments showed that NGINX and Varnish behave this way under their default configuration.

Second, servers may allow for early termination of connections while still caching the response, allowing the attacker to issue GET requests but not read the response, resulting in a similar bandwidth optimization. In our experiments, Squid was the sole server that required the client to consume the entire body before caching an object, while all other caches allowed early termination.

Also note that, while the legitimate traffic received at the website will compete with the attacker's probes for cache space, and result in some bogus objects getting evicted until they can be restored via the TTL refresh loop (Steps 4-5 in the overview, lines 11-19 in Algorithm 5), that does not imply better traffic offload for the origin. Since the cache is still at storage capacity, even this temporary caching of genuinely popular content will result in cache thrashing, and fail to remediate the cache degradation. We demonstrate this effect later in our evaluation, in Section 6.4.

6.3.3 Finding Objects With Imprecise Cache Keys

Identifying a cacheable object with an imprecise cache key definition, ideally a large one, hosted on the target website is an essential part of WCO. Thankfully, this process can be automated using the cache-busting methodology that we previously presented in Chapter 3 to cache bust requests. We adapt and extend that methodology for our purposes here.

Specifically, we implement a recursive web crawler that is seeded by the target web-

site's domain. The crawler issues GET requests for the resources linked from the crawled pages, and checks the response for indications of caching. These include response header heuristics compiled in Chapter 3.

Once the crawl is complete, yielding a list of cacheable objects, we sort these by object size in descending order, and work through the list to automatically reverse engineer their cache key definitions. This process involves crafting a GET request for that object, systematically modifying the frequently keyed HTTP elements in isolation, sending the mutated request, and checking the response for cache status indicators using the aforementioned header heuristics. In particular, we test the following, in the given order:

1 Modify the query string.

2 Include and modify the headers drawn from keyed fields observed in default configurations of popular cache technologies (i.e., Origin, User-Agent, X-Forwarded-Host, X-Forwarded-For, X-Method-Override, Accept, Accept-Language, Accept-Encoding, X-Forwarded-Scheme).

3 Edit the values of headers specified with the `Vary` header.

4 Include `Cache-Control: no-cache`.

5 Add arbitrary cookies.

When we find one object for which we observe a cache miss that can be induced by these attempts, we can stop the process. Detecting one object impacted by an imprecise cache key issue is sufficient for the attack.

This is depicted on line 1 of Algorithm 5 with the routine *findObjectWithImpreciseKey*, which returns both the path for the exploitable object, and the imprecise cache key definition, allowing us to generate an arbitrary number of cache busters by varying the appropriate keyed HTTP element.

6.3.4 Detecting When The Cache Is Full

Finally, an accurate approximation of when the cache is full and the attack is at its peak is an important signal for the attacker. On the one hand, over-approximating the number of redundant objects to cache is costly, as generating new cache busters and sending GET requests for them incur a higher cost than the TTL maintenance loop of sending HEAD requests, providing no real benefit in the process. On the other hand,

under-approximating this number means that the attacker switches to the maintenance loop too early, leaving space in the cache for genuinely popular objects, and hence under-utilizing WCO. We propose two methods to tackle this problem.

First, if the cache capacity can be accurately estimated, the solution is trivial since the stored object size is also known; (*cache capacity / object size*) is the approximate number of objects that the attacker must store. Naturally, this information may not be known for most proprietary deployments; however, also note that many production systems rely on pre-packaged software with baked-in defaults (e.g., caching proxies deployed via containers in public registries, managed cache components provided by hyperscalers like AWS, Azure, GCP) making informed estimations practical if contextual clues for the deployment setup exist. We use a similar methodology in our evaluation, in Section 6.4 to collect information on commonly seen cache sizes.

Second, when cache capacity estimation is not feasible, attackers can perform their own cache hit rate monitoring. Specifically, one can select a highly popular cacheable object (e.g., an image banner on the homepage), periodically issue requests for that object while the attack is in progress, and monitor the cache hits and misses. Observing a cache miss for an object served from the homepage is unlikely and could alone indicate that the attack has escalated to high severity. The attacker can then decide whether to cache more objects or switch to TTL maintenance depending on the cache hit/miss rate calculated over these monitoring probes.

6.4 Evaluation

We now evaluate how WCO performs in practice through a set of detailed experiments.

Unfortunately, testing WCO on real-life websites is not feasible. DoS attacks are explicitly prohibited on all major bug bounty platforms. Even if testing is throttled in an attempt to limit damage, volumetric attacks have real performance and cost consequences for website operators.

Therefore, we perform the bulk of our evaluation ethically in a controlled lab environment, by attacking our own test infrastructure, without disruption to or other negative implications on other parties. This is aligned with all the cache pollution works we cited in Section 6.1, and is the overall standard approach in DoS research. To ensure our evaluation methodology reflects reality, we first perform a set of Internet measurements and then design the rest of the experiments guided by real-life observations.

Table 6.1: Effectiveness of cache-busting techniques on large objects with imprecise keys. Percentages are calculated over the total number of websites with cacheable objects, 4000.

Cache-buster	# of Objects	
Query string	2672	(66.8%)
Accept-Encoding	852	(21.3%)
User-Agent	53	(1.3%)
X-Forwarded-For	25	(0.6%)
Accept	183	(4.6%)
Origin	1778	(44.5%)
X-Method-Override	1398	(34.9%)
X-Forwarded-Scheme	1379	(34.5%)
X-Forwarded-Host	1329	(33.2%)
Accept-Language	102	(2.5%)
Cookie	61	(1.5%)
Vary	363	(9.1%)
Cache-control	25	(0.6%)

6.4.1 Imprecise Cache Keys And Object Size In The Wild

Identifying cacheable objects with imprecise cache keys is the prerequisite for WCO. Therefore, we first perform a large-scale Internet experiment to determine whether imprecise cache keys indeed exist in popular websites. While doing so, we also measure the sizes of these cache-bustable objects, as this is a critical parameter that influences the viability of WCO that we must evaluate later.

We conduct this measurement on the Tranco top 10k domain list generated on April 7, 2025 [99]. [1] We follow the same methodology discussed in Section 6.3.3, where we crawl each website, identify cacheable objects, and then automatically reverse engineer their cache keys. We limit the crawl to a maximum of 10 pages on each of at most 10 unique subdomains per website, in order to avoid generating unreasonable traffic volumes. During cache key reversing, we process objects by size in descending order, and stop the experiment when we find the largest object with an imprecise cache key. Our objective is not to perform a comprehensive measurement study detailing all exploitable objects, but to demonstrate the existence of at least one such object on a website, therefore minimizing the experiment's burden on production systems.

Out of the 10k websites in our dataset, we were able to crawl 5437, while the rest did not respond to HTTP either due to infrastructure issues, or potentially because

[1] Available at https://tranco-list.eu/list/LJL44.

of bot detection defenses. Among the websites successfully tested, we found 4000 that contained cacheable objects, and **3600** of these contained at least one exploitable object with an imprecise cache key. We were unable to detect an imprecise cache key for any object on the remaining 400 websites. [2] Table 6.1 further breaks down the vulnerable keyed elements in the imprecise cache keys for these 3600 objects. The five-number summary for the exploitable objects is as follows: $Min = 2\,\text{bytes}, Q1 = 0.12\,\text{MB}, Median = 0.37\,\text{MB}, Q3 = 1.02\,\text{MB}, Max = 187.54\,\text{MB}$.

We note that these numbers are lower bounds due to the limits we imposed on the crawl due to ethical considerations (e.g., we did not make attempts to bypass bot detection techniques, which are commonly utilized for big file downloads). Case in point, our experiment did not catch a 6 GB operating system image hosted on one of the tested sites, which we later manually confirmed to have an imprecise cache key.

To empirically demonstrate this lower bound, we randomly sampled 100 websites from the subset of sites that presented exploitable objects smaller than the median size, and manually searched each site for larger exploitable objects. 16 sites redirected to other domains and were therefore discarded. On 72 sites, we found larger exploitable objects, and on only 12 sites, we did not find any. Following is a five-number summary of the size increase ratios between the largest manually discovered exploitable object and the previously automatically detected one: $Min = 2.96, Q1 = 43.84, Median = 148.53, Q3 = 1605.43, Max = 59507.35$. This indicates that in many cases, significantly larger cacheable objects with imprecise cache keys exist on the same website.

Figure 6.3 shows the object sizes, where websites are organized into buckets corresponding to the size of the largest object on them (in blue), and the largest object with an exploitable imprecise cache key (in orange).

This experiment confirms that imprecise cache keys are indeed very common, and they can be automatically detected. As for the object size distribution, we confirm that more than 25% of websites have exploitable objects larger than 1 MB, extending into the 100 MB range. However, the median is fairly small at 0.37 MB. We leverage the insights from this measurement in the rest of our evaluation, and test how these object sizes influence the success of WCO.

6.4.2 Cache Capacity In The Wild

To establish realistic cache capacity values to use in our experiments, we selected five popular open-source caching proxy technologies (i.e., Apache Traffic Server (ATS),

[2] We triple checked that 4000, 3600, and 400 are indeed accurate results!

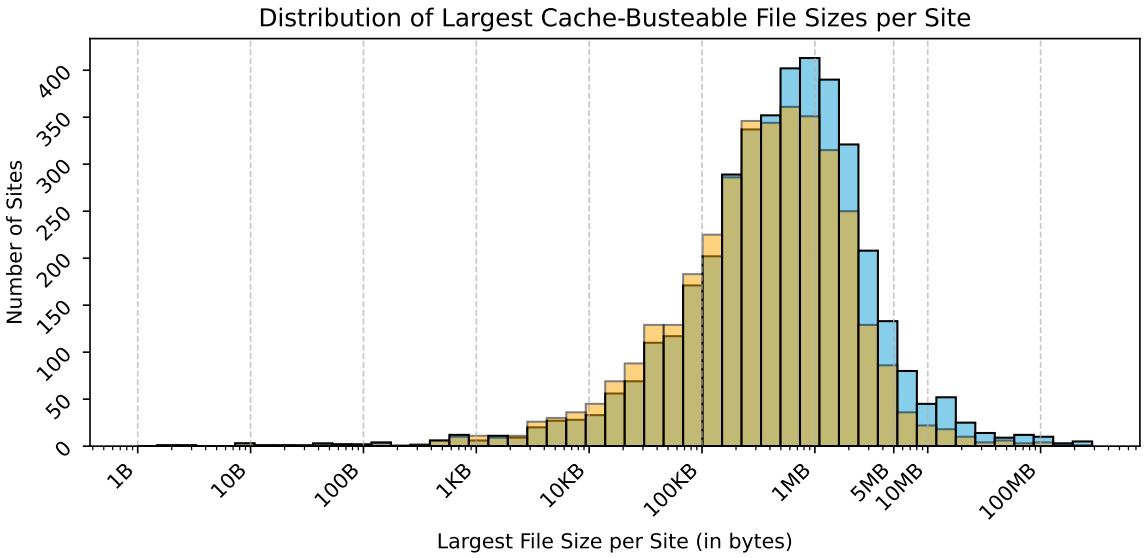

Figure 6.3: Websites grouped into buckets based on the largest detected object size (in blue), and the largest object with an imprecise cache key (in orange).

Table 6.2: Cache sizes observed in configurations on GitHub. "#" denotes the number of configurations we analyzed.

Proxy	#	Min (MB)	Q1 (MB)	Median (MB)	Q3 (MB)	Max (MB)
ATS	23	144	256	256	10240	1048576
HAProxy	34	4	20	164	512	4095
NGINX	29	73	256	1024	1024	51200
Squid	21	16	100	512	5000	51200
Varnish	20	1	256	256	1280	32768

HAProxy, NGINX, Squid, and Varnish), and searched GitHub for projects that use them via the GitHub API. We then analysed the cache configurations these projects are bundled with.

This process yielded 127 projects with valid cache configurations–a sample list is available in Appendix B. Table 6.2 summarizes the results, showing that the median cache store capacity does not exceed 1 GB for any configuration. While the vast majority of caches were below 50 GB, we found a few outliers for ATS that exceeded 100 GB and maxed out at 1 TB. We designed the rest of the experiments in this section in light of these findings.

6.4.3 WCO With Common Defaults

We perform our first WCO experiment with the same selection of five caching proxies as above. This is an initial attack feasibility validation, using the common values we draw from the above measurements for cache configuration parameters.

We configure all proxies to cache every response. We work with objects of size 1 MB, and we set the cache capacity to 50 GB, both informed by our observations in the previous experiment. The only exception is HAProxy, for which the maximum cache capacity is 4095. Other configuration parameters are left at their default settings. In particular, all caches employ the Least Recently Used (LRU) eviction algorithm, and they define their cache keys to include the query string, allowing us to perform cache busting by modifying its value.

To simulate the routine traffic a website receives, we use Web Polygraph, a benchmarking tool for caching proxies and other web intermediaries [105]. Web Polygraph provides a client component that generates the traffic, and a server component that responds to that traffic. We can then place the tested cache in between, measuring metrics such as the cache hit and response time.

Web Polygraph provides predefined workloads simulating common traffic patterns. We use the workload `simple.pg`.[3] This workload represents a mix of repeated and non-repeated requests with a configurable request rate, which directly affects the cache hit rate during the experiment. By default, the recurrence rate is set to 55%, meaning that Web Polygraph requests the same resources 55% of the time. Since we set all responses to be cacheable, we anticipate a consistent hit ratio of around 55%. We configure Web Polygraph to issue 100 requests per second.

We test each cache in isolation by deploying it as a reverse proxy in our lab environment, in front of our origin server that serves static files of varying sizes. The attacker's requests are routed to this origin, while the simulated routine traffic is directed to Web Polygraph's server. The two routes share the same cache. As a result, the impact of WCO manifests in Web Polygraph's benchmarks.

For each experiment, we first run Web Polygraph for 10 minutes without performing any attack to warm the cache. We then run WCO for the next 30 minutes. During both phases, we record the cache hit rate and the mean response time. Compared to the subsequent experiments, we use a longer attack duration here to clearly illustrate the attack's warm-up and maintenance phases.

Table 6.3 presents the results, highlighting the average cache hit rate and mean

[3] `https://github.com/albertok/web-polygraph/blob/master/workloads/simple.pg`

Table 6.3: WCO impact. "HR" stands for Hit Rate, and "RT" for Response Time. Values are averages over the 10-minute phases before and during WCO.

	Before		During	
Cache	HR	RT (ms)	HR	RT (ms)
ATS	55.1	1.1	6.2	13.3
HAProxy	55.1	0.0	0.9	0.8
NGINX	55.1	0.0	2.2	15.0
Squid	55.1	1.0	6.7	1471.2
Varnish	55.1	0.0	0.6	0.0

response time for each cache technology at each experiment phase. The results show that WCO significantly degrades performance in all cases. The cache hit rate drops towards 0%, and in turn, the mean response time increases with the origin server's traffic load. Figure 6.4 visually depicts the cache hit rate drop as the attack progresses.

The results so far are promising, demonstrating that WCO renders the cache ineffective under widely used default configurations.

6.4.4 WCO With Varying Parameters

We next perform a set of experiments to see how cache capacity and cached object size influence the outcome. Having shown that WCO is agnostic to the cache technology implementation, we perform the remaining tests using NGINX as our choice of caching reverse proxy.

In these experiments, we also measure bandwidth and calculate the Damage Ratio (DR) as the parameters vary, in order to realistically validate that WCO is a practical DoS attack. During attack maintenance, we throttle our attack traffic rate to complete a full TTL refresh cycle within 60 seconds (i.e., every cached duplicate receives a HEAD request every 60 seconds approximately), rather than sending requests as quickly as possible. This helps achieve a higher DR to better illustrate the attack's viability. We note that our choice of 60 seconds is extremely conservative; cache TTLs for popular objects can range from hours to days in real-life applications.

Cache Capacity

We run four tests with NGINX configured with a cache storage capacity ranging from 1 GB to 100 GB, informed by our previous measurement experiment. We also repeated the attack using several objects of different sizes in our exploratory studies, ranging

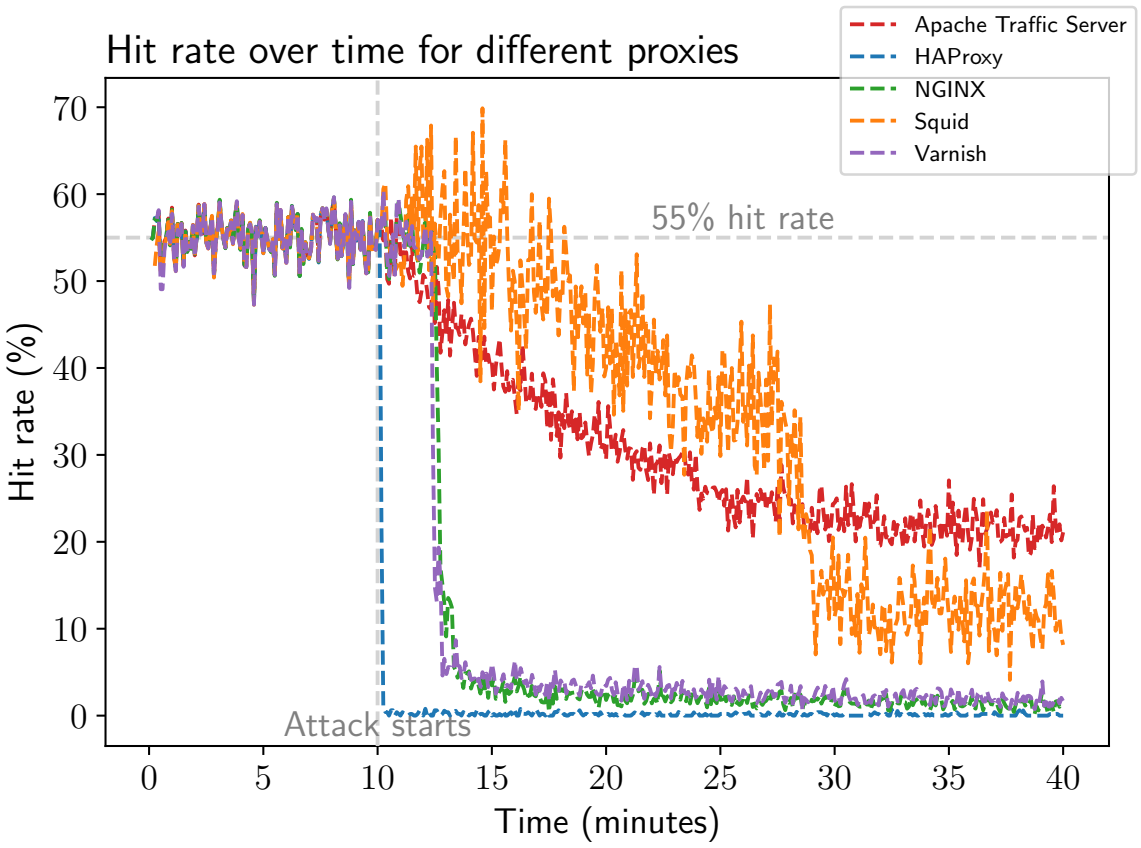

Figure 6.4: Cache hit rate before and during WCO with different caching technologies. The cache size is 50 GB, while the object size is 1 MB.

from 100 KB to 500 MB; here, we present the results with a 1 MB file.

Figure 6.5 shows the cache hit rate before and during the attack, once again demonstrating that WCO rapidly degrades performance in the same way, regardless of the cache capacity. However, we also see in Table 6.4 that the cache capacity has a notable impact on the attacker's cost. Bigger caches require more objects to fill, which in turn translates to higher bandwidth. We stress that this is a conservative case. As we show in the next experiment, the attacker could employ bigger objects to perform the attack efficiently on a 100 GB cache. We remind that the attacker does not need to download the full body of each response, as explained in Section 6.3, which is why the bandwidth values are significantly lower than the cache capacity.

Table 6.4: WCO cost with an object size of 1 MB and varying cache capacity.

Cache Capacity (GB)	T_{before} (MB)	T_{during} (MB)	T_{WCO} (MB)	DR
100	351.37	678.25	153.77	2.13
50	348.42	684.02	133.55	2.51
10	349.98	707.79	36.05	9.92
1	348.76	618.35	3.64	74.05

Table 6.5: WCO cost with a cache capacity of 50 GB and varying object sizes.

Cache Capacity (GB)	T_{before} (MB)	T_{during} (MB)	T_{WCO} (MB)	DR
500	357.82	676.78	15.31	20.84
10	351.92	583.28	7.76	29.83
1	348.42	684.02	133.55	2.51
0.1	350.65	571.81	391.43	0.57

Object Size

We now configure NGINX with a cache capacity of 50 GB and perform WCO using five different object sizes, ranging from 100 KB to 500 MB.

Figure 6.6 shows the resulting cache hit rate, demonstrating that WCO works regardless of object size. Table 6.5 confirms our intuition that larger objects yield far better DR values, but even with a 1 MB object, WCO remains viable.

6.4.5 Eviction Algorithms

LRU is the most common, and often the only supported eviction algorithm, with many web cache technologies. Yet, some of the works we cited in Section 6.1 specifically explore the use of other caching strategies in their defences against cache pollution (e.g., [69]). Therefore, we also evaluate the impact of different eviction algorithms on the effectiveness of WCO.

For this particular experiment, we use Squid instead of NGINX, since only Squid supports the variety of eviction algorithms we would like to test. We set the cache capacity to 50 GB, and the cached object size to 1 MB. We then run tests for each eviction algorithm that Squid supports: LRU, heap LFU, heap GDSF, and heap LFUDA.

Table 6.6 presents the results of this experiment. In short, we saw no difference of note between these different eviction algorithms, and we conclude that WCO is robust

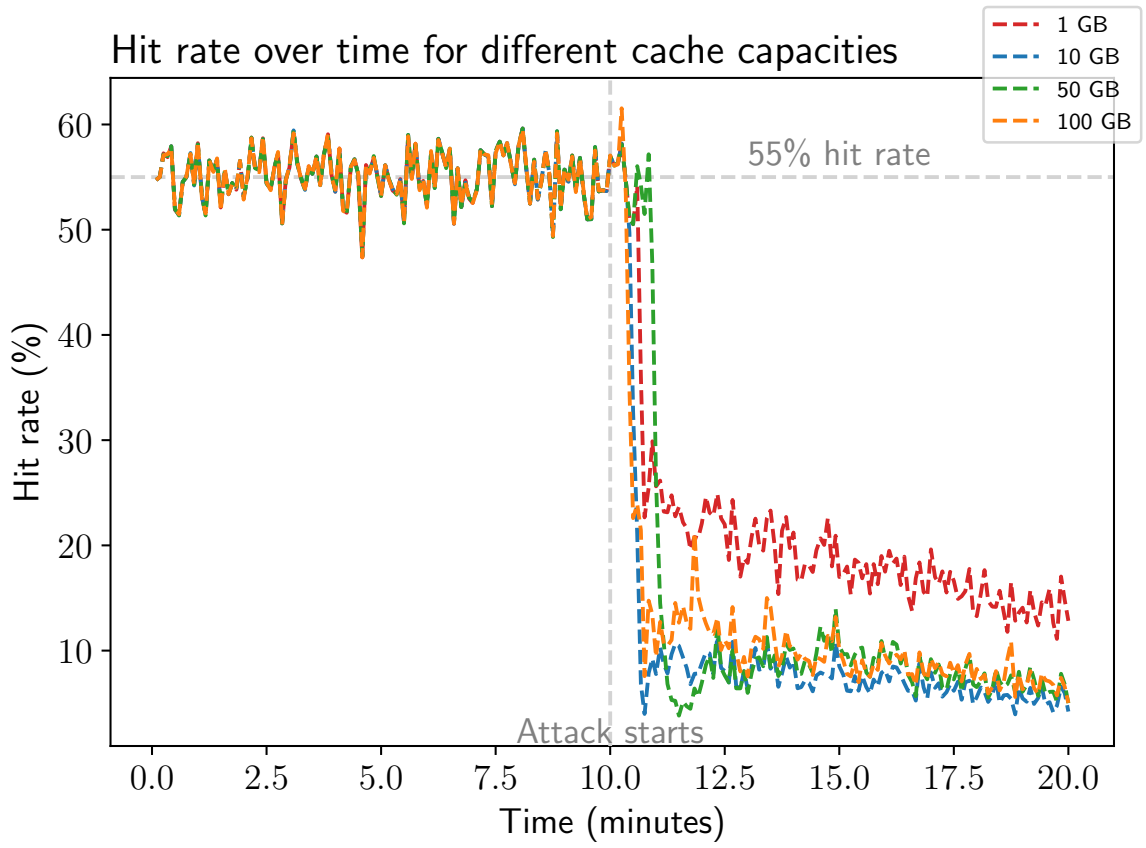

Figure 6.5: Cache hit rate before and during WCO with NGINX for various cache capacities and an object of 1 MB.

against tweaks to the caching strategy. This confirms our previous claim that cache eviction details are irrelevant for the WCO's abuse mechanism of filling a cache to capacity.

6.4.6 WCO-facilitated Cache Poisoning

As we previously pointed out, instead of using WCO for DoS purposes, an attacker can also utilize it as a cache purge mechanism to facilitate other attacks. In this section, we demonstrate this capability by implementing a proof-of-concept cache poisoning attack against a vulnerable web application.

We add to our experiment setup a web application containing a reflected XSS vulnerability, served from the origin. We experiment with different cache capacities, and employ a 1 MB file for the attack. In this experiment, we employ smaller caches because we found that with larger caches, the attack takes an impractically long time to

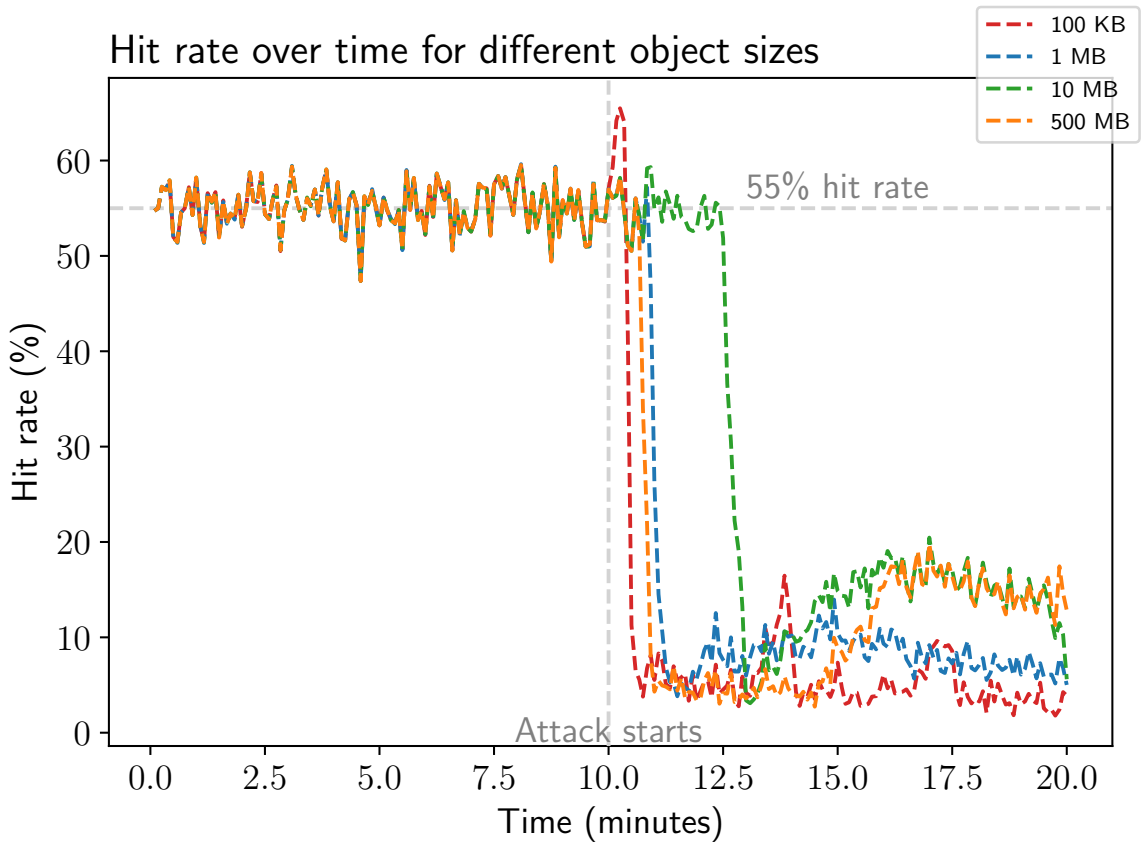

Figure 6.6: Cache hit rate before and during WCO with NGINX for different file sizes and cache capacity of 50 GB.

succeed. Attackers with more resources or larger objects could overcome this limitation and poison larger caches.

We start the test in a state where a response for the vulnerable endpoint is already cached, as it would be in a real-world scenario. The test scenario is as follows. The vulnerable endpoint regularly receives requests from arbitrary clients. The attacker starts a WCO attack, which would eventually evict the legitimate object. At that moment, the attacker would have a short window of opportunity to exploit the XSS and cache the resulting malicious response over the legitimate endpoint. This opportunity is lost if the server receives a request for the endpoint before the attacker can act, as that would once again restore the legitimate object to the cache, and the attacker would need to sustain WCO and wait for that object to get evicted again.

We let this scenario play out with simulated traffic representing arbitrary clients. The attacker performs WCO as usual, while simultaneously trying to poison the vulnerable endpoint, competing with regular traffic. We experiment with various routine

Table 6.6: WCO impact on Squid with different eviction algorithms.

	Before		During	
Eviction Algorithm	HR	RT (ms)	HR	RT (ms)
Heap LRU	55.3	0.7	36.7	375.4
LRU	55.3	0.7	34.2	364.6
Heap GDSF	55.3	0.7	35.2	412.3
Heap LFUDA	55.3	0.7	35.2	381.8

Table 6.7: Average time required to successfully perform cache poisoning by leveraging WCO to purge a target object with different cache capacities and varying regular traffic rates. "-" means the attack did not succeed in a 15 minutes timeout.

Request Rate (r/s)	Average Time to success (s)		
(req/s)	1024 MB	2048 MB	4096 MB
1	32.0	49.0	15.8
10	30.2	50.0	-
50	18.0	63.8	-
100	92.6	108.4	-

request rates for the endpoint, and we perform 5 trials for each of these experiments to measure the average time for a successful poisoning attack.

Table 6.7 shows the results. As the endpoint receives traffic at higher rates, the attacker's window of opportunity for finding the legitimate object evicted gets smaller. Despite that, the attack succeeds within minutes. In practice, this difference in timing is likely of no consequence for the attacker, as long as the cache can be poisoned within a reasonable amount of time. We conclude that WCO may indeed provide the attacker with the capability to purge specific objects of interest from a cache on command, and facilitate exploitation of previously immaterial vulnerabilities.

6.4.7 Summary of Results and Limitations

Our measurements over Tranco top 10k show that more than one third of these popular websites have cacheable objects with imprecise cache keys, confirming the premise for WCO.

The remaining experiments demonstrate that the core mechanism of WCO, that is, filling caches with duplicate objects, forcing the eviction of everything else, and leaving the cache in a perpetual state of cache thrashing, is viable with all tested caching proxies, regardless of the parameters such as cache capacity and exploitable object size.

These parameters, however, do have practical impacts on whether WCO is viable as a DoS vector. Extreme values (very small files, very large caches) can become cost-prohibitive, resulting in low DR values. This is an inherent limitation of the mechanism and methodology we propose in this research, and there are cases where WCO may not be an effective attack.

However, as also evidenced by our study of commonly seen object sizes and cache capacity configurations in the wild, such extreme values for these parameters are not the norm, and there are many opportunities to launch viable WCO attacks in realistic web application deployment settings.

6.5 Mitigations

We now describe a number of mitigation strategies for WCO, and test their effectiveness.

6.5.1 Cache Deduplication

An effective mitigation against WCO is deduplication. Cache deduplication may be implemented by computing checksums of cacheable responses as they are received, comparing them against the checksums of already cached objects, and only storing a new copy if the object is unique in storage. Otherwise, a data structure would be updated to associate the different caches with a single stored copy of the object.

While deduplication is a well-known technique designed for this exact problem, its application to web caches could be challenging. Both the checksum computation and the cache key tracking data structure management tasks can get prohibitively costly for a busy server, and worse, cause processing delays, increasing application response times.

We run a simple experiment to estimate this cost by checksumming files of various sizes and measuring the time the operation takes. We test a cryptographic hash (i.e., SHA512), and also a non-cryptographic checksum (xxHash, specifically the XXH3 variant [142]) designed to be very fast to compare. We perform 1000 trials for every experiment, and present the average times elapsed.

Table 6.8 summarizes the results. This may appear negligible in isolation, but given that today's production platforms process millions of requests per second (e.g., [7]), processing cycles and latency may quickly become unreasonable.

Table 6.8: Average CPU time over 1000 runs to compute file hashes and checksums.

File Size	xxHash Avg. CPU Time (s)	SHA 512 Avg. CPU Time (s)
500 MB	0.24258	0.81138
10 MB	0.00548	0.02170
5 MB	0.00288	0.01296
1 MB	0.0007	0.00304
500 KB	0.0004	0.00161
100 KB	0.00013	0.00042

6.5.2 Anomaly Detection

Anomaly detection is the main defence approach adopted by nearly all cache pollution defence works we referenced in Section 6.1. Most relevant for our context, the works by Gao et al. and Deng et al. that present cache pollution attacks and defences on forward web caches propose a detection scheme based on various request and traffic features [48, 30]. These works show that their approach is effective, but it also comes with processing overhead and false positives. Moreover, it can take up to 10 hours to catch anomalies. These downsides are not unique to the cited works, and are expected of any anomaly detection approach to security.

WCO can also be detected in this manner, likely more easily than other cache pollution attacks, due to its noisier nature. If the shortcomings of anomaly detection are acceptable, this could be a viable solution. However, we also expect the costs to be material for a real-life application, and as discussed in the cited works, IP blocking capabilities themselves may be weaponized by an attacker to block NAT gateways. All in all, we do not believe anomaly detection should be the preferred approach if a more effective solution can be found.

6.5.3 Rate Limits

WCO generates a large volume of traffic sustained over a long time for the TTL maintenance loop. Therefore, we explore rate limiting separately from more advanced anomaly detection as a less costly option. We conduct an experiment with the same evaluation setup to test this approach. We use NGINX, set the cache capacity to 50 GB, and employ a 1 MB file as the cached object. We configure Web Polygraph to issue 10 requests per second from 10 different IP addresses, for a total of 100 requests per second. We run Web Polygraph for 10 minutes without performing any attack, and then run

Table 6.9: WCO cost with different rate limits, a cache of 50 GB and an object of 1 MB.

Rate Limit (req/s)	Before		During	
	HR	RT (ms)	HR	RT (ms)
50	55.1	0.0	54.9	0.0
10	54.6	0.0	54.6	0.0
5	38.4	0.0	37.8	0.0
1	11.7	0.2	10.1	0.7

together with WCO for another 10 minutes. In both phases, we measure the hit rate and the mean response time of the cache. We repeat this experiment while applying various rate limits at the cache, ranging from 1 to 50 requests per second. We tune the attack intensity to match the rate limit in each experiment. If the attacker issues requests faster than the configured limit, the rate limiter will begin dropping excess requests. Those drops will discard many of the attacker's requests and prevent the sustained traffic required to fill the cache. Conversely, an attacker that reduces its rate to be at or below the measured limit can avoid being dropped and, although the attack proceeds more slowly, given sufficient time, it can still gradually populate the cache. In our experiments, we calibrate the attack rate to the applied limit. In the real world, an attacker could easily spot and measure the rate limit and adjust accordingly.

Table 6.9 shows the cache hit rate before and during the attack. The results indicate that rate limiting is effective in reducing the attack's impact. As expected, the more restrictive the limit, the higher the cache hit rate during the attack. That said, rate limits below 10 requests per second, while being effective in reducing the impact of the attack, naturally also impact regular traffic, causing a drop in the cache hit rate before the attack begins. This once again demonstrates that rate limits cannot be applied generally to all traffic without hurting normal traffic flows, and just like anomaly detection, false positives can be disruptive, or they can even be weaponized by an attacker to get legitimate clients blocked.

6.5.4 Stricter Cache-Key Design

Abuse of imprecise cache keys is a necessary precondition for WCO. That also makes eliminating imprecise cache keys the optimal mitigation. In light of all the conceptualization, discussion, and experiments we presented in this work, we argue that this is the preferred approach. The main drawback of this approach is that it requires website operators to thoroughly analyze their cache configurations and apply the necessary

corrections. At the same time, the same imprecise cache detection methodology we presented in Section 6.3.3 can be repurposed to run deep crawls on websites and identify all problem cache keys, automating the bulk of the work. Revising the cache key still requires manual work informed by contextual knowledge of the surrounding application, but we argue that this cost is preferable over the challenges of sifting through false positives or suffering a sustained performance overhead.

Case in point, we developed a penetration testing tool that automates the systematic detection of imprecise cache key configurations on websites. The tool functions as a web crawler that recursively explores a target site, collecting and testing all accessible resources. For each resource, it issues a series of HTTP requests, each differing from the baseline request by a random modification to one element; specifically, the query string, selected headers, or cookies. The tool then applies response header heuristics derived from the work of Mirheidari et al. [88] to determine whether a given modification caused the request to bypass the cache. If a modified request results in a cache miss while the corresponding response remains identical to the unmodified version, the tool infers that the cache key configuration is imprecise. The tool provides recommendations to the user on how to patch its findings using natural language. The tool is based on the methodology discussed in this chapter, and released as open source for researchers and practitioners to use freely in their own environments.

All in all, we assert that the consequences of imprecise cache keys should be communicated to website operators and cache administrators, and crafting strictly precise cache key definitions should be considered a best practice.

6.6 Discussion

6.6.1 Novelty

We now reiterate the differentiating factors between WCO and prior cache pollution strategies. While traditional attacks rely on knowledge of cache access patterns, file popularity distributions, or the specific eviction algorithms employed by the target, our results show that WCO operates entirely without such information. Our attack is solely based on identifying imprecise cache keys: a process that, as demonstrated in Section 6.3 and Section 6.4.1, can be trivially automated. This simplicity makes the attack both easier to deploy and more general in scope, removing the need for any target-specific reconnaissance or fine-tuned strategies. WCO 's actively generates arbitrary cache entries using cache-busting techniques, rather than relying on pre-existing

unpopular resources, removing any constraints imposed by the amount and size of the existing content available on the victim site. WCO is in fact only limited by environmental factors, such as available bandwidth, server-side rate limiting, or the cache expiration policy.

Furthermore, leveraging lightweight HEAD requests instead of full object downloads enables sustained disruption with minimal effort and very low bandwidth and computational cost, making it accessible even to resource-limited adversaries. Finally, the consequences of WCO extend beyond mere cache performance degradation. Its ability to force eviction of arbitrary objects enables new avenues for exploitation, such as cache poisoning or cache-based side channels, by removing the usual precondition that the targeted resource must not already be cached. We discussed how, for instance, an attacker can leverage WCO to create a window of opportunity for turning a reflected XSS vulnerability into a stored one through cache poisoning. Overall, we argue that the results presented in this chapter highlight that WCO is not a minor variation of existing cache attacks, but a fundamentally different threat model.

6.6.2 What About Content Delivery Networks?

In this chapter, we focused our discussion on server-side caches, and in particular for our evaluation, stand-alone caching proxies. This scope is not unrealistic; such caching reverse proxies deployed in multiple layers are ubiquitous and essential components of scalable web and cloud architectures. That said, Content Delivery Networks (CDNs) that operate massively distributed Internet overlays of caching reverse proxies also provide immense amounts of caching capacity to the web.

Unfortunately, we cannot perform experiments with CDNs due to the aforementioned ethics considerations. Regardless of the effectiveness of the attack, there would be a cost to the infrastructure operators, and furthermore, volumetric attack testing is explicitly prohibited in the Acceptable Use Policies of all major CDNs we checked.

Even without testing, we assume that WCO would not work on CDNs. While there are no public records of CDN cache sizes, a quick Internet search reveals many speculative accounts of CDNs deploying caches with capacities that are orders of magnitude bigger than common stand-alone cache configurations. As evidenced by our evaluations, WCO does not yield a viable DR value at such extreme values. Therefore, we operate under the assumption that WCO is not relevant for CDNs, and this is a fundamental limitation of our work.

To confirm this assumption and to notify them of our work, we contacted Akamai,

AWS CloudFront, CDN77, Cloudflare, Fastly, Google Cloud CDN, KeyCDN, Bunny CDN, and OVHCloud. We shared our findings with them by providing a summary of this research, our proof-of-concept attack tools for their testing, and asking for their feedback.

Akamai, Cloudflare, Fastly, and Google all acknowledged the research, but also confirmed that WCO would not be possible at their scale. AWS responded by saying that DoS attacks are out of scope for their bug bounty program (even though we did not seek a bounty), and did not provide further comments. The remaining parties did not respond to contact attempts.

6.6.3 Ethical Considerations

All the attack experiments we present in this chapter were conducted in realistic, but controlled, lab environments, with no ethical implications.

We also designed the Tranco top 10k experiments to minimize the traffic load on the targeted websites and to eliminate all damage outcomes. Specifically, we limited our crawls to at most 10 subdomains for each website, and to visit at most 10 pages per subdomain. We also limited our traffic rate to 1 request per second to avoid overwhelming the servers. To minimize the overhead of testing the websites for imprecise cache keys, we stopped on detecting the first successfully cache-busted object.

Most importantly, we note the following additional ethical considerations. The issue we present is a consequence of imprecise cache key configuration rather than a vendor-specific implementation vulnerability; consequently, disclosure to caching-proxy software vendors was not appropriate or likely to be actionable. This is fundamentally a configuration problem affecting website and proxy operators. We did not perform vulnerability detection on production sites that use caching proxies, which would have required active experiments capable of causing DoS. Conducting such testing would have been unethical and out of scope, even for sites that publish vulnerability disclosure policies, so we refrained from active probing. Therefore, we did not identify or report any specific vulnerable sites.

Finally, we did not issue a separate public advisory because, to our knowledge, no CERT-like organization accepts advisories covering broad configuration-class issues of this scope. Instead, this chapter itself documents the problem, its impact, and recommended mitigations and therefore serves as a public advisory for operators and researchers.

6.7 Chapter Summary

In this chapter, we presented WCO, a novel cache pollution attack that abuses imprecise cache keys to fill a web cache with duplicate objects, forcefully purges other content, and causes severe cache degradation. We demonstrated that the primary impact of WCO is a Denial of Service, but also showed that its ability to purge specific objects on command can be a valuable tool for adversaries to facilitate other attacks, such as cache poisoning. A key novelty of WCO is its simplicity and broad applicability; unlike prior work, it does not require any knowledge of the cache's eviction algorithm or the website's traffic patterns. Instead, it leverages imprecise cache keys: a fundamental misconfiguration in how caches are often set up, making it a threat that can be exploited by unsophisticated attackers.

Our evaluation confirms the viability of WCO in practice. A large-scale measurement study on the Tranco top 10k websites revealed that imprecise cache keys are widespread, with over a third of the popular websites having cacheable objects that could be exploited. Subsequent lab experiments with popular caching proxies like NGINX, Squid, and Varnish, demonstrated that WCO is effective across different technologies, cache capacities, object sizes, and eviction algorithms. We introduced the Damage Ratio (DR) to quantify the attack's efficiency, showing that WCO is a practical DoS vector in many realistic deployment scenarios. We also successfully demonstrated a proof-of-concept cache poisoning attack facilitated by WCO. Our investigation acknowledges the limitations of the attack, concluding that large-scale Content Delivery Networks (CDNs) are likely not vulnerable due to their massive cache capacities, a conclusion supported by feedback from several major CDN providers.

Finally, we explored various potential mitigations. While strategies like cache deduplication, anomaly detection, and rate limiting can be effective to some extent, they all come with significant drawbacks, such as performance overhead, complexity, false positives, or impacting legitimate users. We conclude that the most effective and efficient solution is to address the root cause: imprecise cache keys. By ensuring that every unique cache key maps to a unique object, website operators can eliminate this attack vector entirely. We argue that designing precise cache keys should be considered a security best practice, and believe that raising awareness of the severe security implications of imprecise cache keys is crucial for improving the security and resilience of the web ecosystem.

Chapter 7

Conclusions

In this thesis, we have presented a comprehensive analysis of the impact of security vulnerabilities in web caches. We began by exploring the fundamental role that web caches play in enhancing web performance and user experience. Web caches have become crucial components of the modern web ecosystem, significantly reducing latency and bandwidth consumption. They are indispensable for meeting the high performance expectations of users and ensuring the efficient delivery of web content. Several studies have shown how even small delays in loading times can lead to frustration, users abandoning websites, significantly impacting the profit of online businesses. However, despite their apparent benefits, if not properly configured and managed, web caches also pose an underestimated security risk. Even though some studies have investigated specific vulnerabilities in web caches, we argue that the systemic security of web caches is still an underexplored area. This thesis, while not exhaustive, aims to fill this gap by providing a systematic analysis of the security implications of web cache vulnerabilities.

Contributions We start by discussing different techniques to detect the presence of web caches and to discern between cached and non-cached content in Chapter 3. Identifying cached responses is a prerequisite for any meaningful security assessment of web caches. However, this task is complicated by the lack of standardized headers for conveying cache status and the selective deployment of caches. These techniques are also crucial for developing automated tools that can scan for vulnerabilities at scale. We present multiple approaches to tackle this challenge. The first method leverages cache status headers, such as `X-Cache`, using a fuzzy matching algorithm to account for their inconsistent and non-standardized use. The second method uses timing patterns analysis, eliminating the need for any cache-specific headers, and is therefore applicable

even when such headers are absent or unreliable. We conclude the chapter by exploring alternative detection strategies which are only effective in specific deployment scenarios, but provide insights into new avenues for cache detection. With this, we answer Research Question Q1., demonstrating that it is possible to reliably detect web caches and distinguish cached responses from non-cached ones using a combination of techniques.

Using these techniques as a basis, in Chapter 4, we present a set of vulnerabilities detection techniques designed to automatically and autonomously identify security flaws in web caches, answering Research Question Q2.. We focus on two primary categories of vulnerabilities that have received limited attention in prior research. Moreover, there was a lack of automated tools for their detection, hindering large-scale security assessments. The two categories we focus on are Web Cache Deception (WCD) and Cache Poisoning Denial of Service (CP-DoS). WCD vulnerabilities allow attackers to trick caches into storing sensitive information, which can then be accessed by unauthorized users. CP-DoS vulnerabilities enable attackers to exploit caching mechanisms to render web resources unavailable to legitimate users. We develop a detection technique for CP-DoS vulnerabilities introduced by misconfigurations in the Cross-Origin Resource Sharing (CORS) policy.

We use our vulnerabilities detection techniques to perform large-scale analysis of the prevalence of WCD and CP-DoS vulnerabilities in the wild, uncovering a significant number of vulnerable websites. Specifically, we find 1,180 websites vulnerable to WCD attacks in the Alexa top 10k, and 45 websites vulnerable to CP-DoS attacks among the 6,862 websites from the Alexa top 50k that used CORS. We answer Research Question Q3., providing empirical evidence of the prevalence of web cache vulnerabilities on popular websites.

In Chapter 5, we investigate the broader security implications of web cache vulnerabilities. We demonstrate how these vulnerabilities can be combined with other common web application flaws to create novel attack vectors. Through a series of case studies, we illustrate the potential consequences of such composite attacks, including data leakage, cache poisoning, and denial of service. These case studies highlight the need for a holistic approach to web application security that considers the interactions between caching mechanisms and other components of the web ecosystem. We answer Research Question Q4. by showcasing the significant security impact of web cache vulnerabilities when combined with other web issues, emphasizing the importance of comprehensive security assessments that account for these interactions.

Finally, in Chapter 6, we introduce a new class of vulnerabilities, termed Web Cache Overflow (WCO). WCO vulnerabilities are enabled by imprecise cache key definitions,

allowing attackers to fill a cache with multiple copies of the same resource, each identified by a different cache key, using cache busting techniques. This can lead to increased load on origin servers and caches, resulting in denial of service for legitimate users.

A Systemic Issue All the vulnerabilities and issues discussed in this thesis point to a systemic problem in the design and deployment of web caches. They all share a common characteristic: they are not the result of isolated misconfigurations or implementation flaws in individual web caches, but rather stem from the complex interplay between web caches, web applications, and the broader web ecosystem. The distributed nature of these systems makes detecting these vulnerabilities a challenging task, as they often require a comprehensive understanding of how different components interact with each other. There is no single entity to blame, and there are no unit tests to run, signatures to match, CVEs to track and patches to apply. Even worse, this systemic essence makes addressing and mitigating these vulnerabilities particularly difficult, as it necessitates coordinated efforts across multiple stakeholders.

Moreover, the incentives for individual entities to invest in securing web caches may be misaligned with the broader goal of enhancing web security. For instance, a website owner may prioritize performance improvements over security considerations, leading to configurations that favour caching efficiency at the expense of security. Similarly, cache providers may focus on optimizing cache performance and scalability, potentially overlooking security implications.

Solving these issues remains an open challenge for the research community, as it requires a holistic view that maps the complex interactions between these components.

The findings presented in this thesis tackle this issue by automating the detection of web cache vulnerabilities at scale, removing the need for manual analysis and intervention, and lowering the barrier for widespread security assessments of web caches. But they are only a first step towards a more secure web caching ecosystem.

Future Work In this thesis, we laid the groundwork for understanding and addressing the systemic security issues associated with web caches. However, we focused primarily on detection and prevalence assessment, leaving several avenues for future research. First, there is a need for mitigation strategies to effectively address, and possibly prevent, these kinds of vulnerabilities. Developing best practices, tools, and frameworks that facilitate secure configurations and deployments of web caches is essential, especially considering the challenges involved in coordinating efforts across multiple stakeholders. Researchers could explore ways to improve communication between websites

and web caches regarding caching policies and security requirements, enabling more dynamic and context-aware caching decisions without compromising security.

During our research projects, we frequently observed hesitance from website owners and cache providers to adopt security measures by default, often due to concerns about breaking existing functionalities and legacy systems. A notable example is Cloudflare's "Deception Armor" [24], which effectively mitigates WCD vulnerabilities by matching the file extension in the URL with the actual content type. However, this feature is disabled by default and not widely adopted, likely due to fears of disrupting functionality on legacy systems. Future work could explore strategies to encourage the adoption of such security features, or develop new mechanisms that balance security and compatibility more effectively.

As with other areas of web security, education plays a crucial role in improving the overall security posture of web caches. Many developers and operators are still unaware of the security implications of web caching, or underestimate the risks involved. It is crucial for the research community to continue efforts in spreading awareness and providing resources that help developers understand and mitigate these risks. This could come in the form of guidelines and best practices, educational materials, or even integration of security considerations into web development frameworks and tools.

Finally, as the web ecosystem continues to evolve, new caching technologies and architectures are likely to emerge. Future research should monitor these developments and assess their security implications, ensuring that the lessons learned from current web cache vulnerabilities are applied to new systems. This proactive approach will help prevent the recurrence of similar systemic issues in the future.

Bibliography

[1] Onur Aciiçmez, Werner Schindler, and Çetin K. Koç. Improving brumley and boneh timing attack on unprotected ssl implementations. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 139–146, New York, NY, USA, 2005. Association for Computing Machinery.

[2] Yehuda Afek, Anat Bremler-Barr, Shoham Danino, and Yuval Shavitt. A flushing attack on the DNS cache. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2299–2314, Philadelphia, PA, August 2024. USENIX Association.

[3] Akamai Developer. EdgeWorkers. `https://developer.akamai.com/akamai-edgeworkers-overview`.

[4] Akamai Technologies. Facts & Figures. `https://www.akamai.com/us/en/about/facts-figures.jsp`.

[5] Virgílio Almeida, Daniel Menascé, Rudolf Riedi, Flávia Peligrinelli, Rodrigo Fonseca, and Wagner Meira Jr. Analyzing Web Robots and Their Impact on Caching. In *Workshop on Web Caching and Content Distribution*, 2001.

[6] Apache HTTP Server Project. Caching Guide. `https://httpd.apache.org/docs/2.4/caching.html`.

[7] Jeff Barr. AWS Identity and Access Management. AWS News Blog. `https://aws.amazon.com/blogs/aws/happy-10th-birthday-aws-identity-and-access-management/`.

[8] Adam Barth. The Web Origin Concept. RFC 6454, December 2011.

[9] Daniel J Bernstein. Cache-timing attacks on aes, 2005.

[10] Mike Bishop. HTTP/3. RFC 9114, June 2022.

[11] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 621–628, New York, NY, USA, 2007. Association for Computing Machinery.

[12] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In Vijay Atluri and Claudia Diaz, editors, *Computer Security – ESORICS 2011*, pages 355–371, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[13] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005. Web Security.

[14] BuiltWith. BuiltWith Technology Lookup. `https://trends.builtwith.com/CDN/Content-Delivery-Network`.

[15] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Content security problems? evaluating the effectiveness of content security policy in the wild. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1365–1375, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. CCSP: Controlled relaxation of content security policies by runtime policy composition. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 695–712, Vancouver, BC, August 2017. USENIX Association.

[17] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Semantics-based analysis of content security policy deployment. *ACM Trans. Web*, 12(2), jan 2018.

[18] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. We still Don't have secure Cross-Domain requests: an empirical study of CORS. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1079–1093, Baltimore, MD, August 2018. USENIX Association.

[19] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. Host of Troubles: Multiple Host Ambiguities in HTTP Implementations. In *ACM Conference on Computer and Communications Security*, 2016.

[20] Cloudflare. Creating Cache Keys. `https://support.cloudflare.com/hc/en-us/articles/115003206852s`.

[21] Cloudflare. Origin Cache-Control. `https://support.cloudflare.com/hc/en-us/articles/115003206852s`.

[22] Cloudflare. The Cloudflare Global Anycast Network. `https://www.cloudflare.com/network/`.

[23] Cloudflare. Understanding Cloudflare's CDN, 2021. `https://support.cloudflare.com/hc/en-us/articles/200172516-Understanding-Cloudflare-s-CDN`.

[24] Cloudflare. Cache Deception Armor, 2025. `https://developers.cloudflare.com/cache/cache-security/cache-deception-armor/`.

[25] Cloudflare. Cache Keys, 2025. `https://developers.cloudflare.com/cache/how-to/cache-keys/`.

[26] Cloudflare. Origin Cache-Control, 2025. `https://varnish-cache.org/docs/7.7/users-guide/vcl.html`.

[27] Cloudflare Docs. Cloudflare Workers Documentation, 2021. `https://developers.cloudflare.com/workers/`.

[28] Mauro Conti, Paolo Gasti, and Marco Teoli. A Lightweight Mechanism for Detection of Cache Pollution Attacks in Named Data Networking. *Computer Networks*, 57(16):3178–3191, 2013.

[29] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and limits of remote timing attacks. *ACM Trans. Inf. Syst. Secur.*, 12(3), jan 2009.

[30] Leiwen Deng, Yan Gao, Yan Chen, and Aleksandar Kuzmanovic. Pollution Attacks and Defenses for Internet Caching Systems. *Computer Networks*, 52(5):935–956, 2008.

[31] ATS Developers. HTTP Proxy Caching — Apache Traffic Server 7.1.11 documentation.

[32] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Smart Card Research and Applications*, pages 167–182, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[33] Akamai Documentation. Caching, 2021. `https://learn.akamai.com/en-us/webhelp/api-gateway/api-gateway-user-guide/GUID-B717E657-4C07-4B76-934A-36F1C40F91AE.html`.

[34] Fastly Documentation. Configuring Caching, 2020. `https://docs.fastly.com/en/guides/configuring-caching`.

[35] Adam Doupé, Weidong Cui, Mariusz H. Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. Dedacota: Toward preventing server-side xss via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 1205–1216, New York, NY, USA, 2013. Association for Computing Machinery.

[36] Fastly. Compute@Edge. `https://www.fastly.com/products/edge-compute/use-cases`.

[37] Fastly. Date | Fastly Documentation.

[38] Fastly. Fastly Developer Hub – X-Cache. `https://developer.fastly.com/reference/http-headers/X-Cache/`.

[39] Fastly. Fastly Network Map. `https://www.fastly.com/network-map`.

[40] Mattia Fazzini, Prateek Saxena, and Alessandro Orso. Autocsp: Automatically retrofitting csp to web applications. In *IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 336–346, Florence, Italy, 2015. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering.

[41] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, page 25–32, New York, NY, USA, 2000. Association for Computing Machinery.

[42] Roy T. Fielding, Mark Nottingham, and Julian Reschke. HTTP Caching. RFC 9111, June 2022.

[43] Roy T. Fielding, Mark Nottingham, and Julian Reschke. HTTP Semantics. RFC 9110, June 2022.

[44] Roy T. Fielding, Mark Nottingham, and Julian Reschke. HTTP/1.1. RFC 9112, June 2022.

[45] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, June 2014.

[46] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014.

[47] Zeke Gabrielse. Tell HN: Cloudflare is rewriting my Date response headers | Hacker News.

[48] Yan Gao, Leiwen Deng, Aleksandar Kuzmanovic, and Yan Chen. Internet Cache Pollution Attacks and Countermeasures. In *IEEE International Conference on Network Protocols*, 2006.

[49] Nethanel Gelernter and Amir Herzberg. Cross-site search attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1394–1405, New York, NY, USA, 2015. Association for Computing Machinery.

[50] Omer Gil. Web Cache Deception Attack. Black Hat USA, 2017. `https://www.blackhat.com/us-17/briefings.html#web-cache-deception-attack`.

[51] Omer Gil. Web Cache Deception Attack, 2017. `https://omergil.blogspot.com/2017/02/web-cache-deception-attack.html`.

[52] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless timing attacks: Exploiting concurrency to leak secrets over remote connections. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1985–2002. USENIX Association, August 2020.

[53] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless timing attacks: Exploiting concurrency to leak secrets over remote connections. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1985–2002. USENIX Association, August 2020.

[54] Yaron Y. Goland. End-To-End Confusion – The Changing Meaning of End-To-End in Transport and Application Protocols – Stuff Yaron Finds Interesting, November 2005.

[55] Yaron Y. Goland. SOA and the End-To-End Morass, November 2005.

[56] Matteo Golinelli, Elham Arshad, Dmytro Kashchuk, and Bruno Crispo. Mind the CORS. In *EEE International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS)*, November 2023.

[57] Run Guo, Jianjun Chen, Baojun Liu, Jia Zhang, Chao Zhang, Haixin Duan, Tao Wan, Jian Jiang, Shuang Hao, and Yaoqi Jia. Abusing CDNs for Fun and Profit: Security Issues in CDNs' Origin Validation. In *IEEE International Symposium on Reliable Distributed Systems*, 2018.

[58] Abdelhak Hidouri, Mohamed Hadded, Nasreddine Hajlaoui, Haifa Touati, and Paul Muhlethaler. Cache Pollution Attacks in the NDN Architecture: Impact and Analysis. In *International Conference on Software, Telecommunications and Computer Networks*, 2021.

[59] Abdelhak Hidouri, Haifa Touati, Mohamed Hadded, Nasreddine Hajlaoui, Paul Muhlethaler, and Samia Bouzefrane. Q-ICAN: A Q-learning Based Cache Pollution Attack Mitigation Approach for Named Data Networking. *Computer Networks*, 235, 2023.

[60] Geoff Huston. The Middleware Dilemma, 2001.

[61] Geoff Huston. The middleware muddle. *Internet Protocol Journal*, 4(2):22–27, 2001.

[62] Evan J. Misconfigured cors, 2016.

[63] Bahruz Jabiyev, Anthony Gavazzi, Kaan Onarlioglu, and Engin Kirda. Gudifu: Guided differential fuzzing for http request parsing discrepancies. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 235–247, 2024.

[64] Bahruz Jabiyev, Steven Sprecher, Anthony Gavazzi, Tommaso Innocenti, Kaan Onarlioglu, and Engin Kirda. FRAMESHIFTER: Security Implications of HTTP/2-to-HTTP/1 Conversion Anomalies. In *USENIX Security Symposium*, August 2022.

[65] Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin Kirda. T-reqs: Http request smuggling with differential fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1805–1820, New York, NY, USA, 2021. Association for Computing Machinery.

[66] Yaoqi Jia, Yue Chen, Xinshu Dong, Prateek Saxena, Jian Mao, and Zhenkai Liang. Man-in-the-browser-cache: Persisting https attacks via browser cache poisoning. *computers & security*, 55:62–80, 2015.

[67] Yaoqi Jia, Xinshu Dong, Zhenkai Liang, and Prateek Saxena. I know where you've been: Geo-inference attacks via the browser cache. *IEEE Internet Computing*, 19(1):44–53, 2015.

[68] Martin Johns. Script-templates for the content security policy. *J. Inf. Secur. Appl.*, 19(3):209–223, jul 2014.

[69] Amin Karami and Manel Guerrero-Zapata. An ANFIS-based Cache Replacement Method for Mitigating Cache Pollution Attacks in Named Data Networking. *Computer Networks*, 80:51–65, 2015.

[70] Christoph Kerschbaumer., Sid Stamm., and Stefan Brunthaler. Injecting csp for fun and security. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy - ICISSP,*, pages 15–25, Online, 2016. INSTICC, SciTePress.

[71] Kettle. HTTP Desync Attacks: Request Smuggling Reborn, August 2019.

[72] James Kettle. Exploiting cors misconfigurations for bitcoins and bounties, 2016.

[73] James Kettle. Practical Web Cache Poisoning. PortSwigger Web Security Blog, 2018. `https://portswigger.net/blog/practical-web-cache-poisoning`.

[74] James Kettle. Breaking the chains on HTTP Request Smuggler, December 2019.

[75] James Kettle. HTTP Desync Attacks: what happened next, October 2019.

[76] James Kettle. Web Cache Entanglement: Novel Pathways to Poisoning. PortSwigger Research, 2020. `https://portswigger.net/research/web-cache-entanglement`.

[77] James Kettle. HTTP/2: The Sequel is Always Worse, August 2021.

[78] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[79] Tobias Lauinger. Security & Scalability of Content-Centric Networking. Master's thesis, Technische Universität Darmstadt, 2010.

[80] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A. Vela Nava, and Martin Johns. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1709–1723, New York, NY, USA, 2017. Association for Computing Machinery.

[81] Emil Lerner. http2smugl, September 2022.

[82] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You've Got Vulnerability: Exploring Effective Vulnerability Notifications. In *USENIX Security Symposium*, 2016.

[83] Yuejia Liang, Jianjun Chen, Run Guo, Kaiwen Shen, Hui Jiang, Man Hou, Yue Yu, and Haixin Duan. Internet's invisible enemy: Detecting and measuring web cache poisoning in the wild. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 452–466, 2024.

[84] Chaim Linhart, Amit Klein, Ronen Heled, and Steve Orrin. Http request smuggling, 2005.

[85] Dapeng Man, Yongjia Mu, Jiafei Guo, Wu Yang, Jiguang Lv, and Wei Wang. Cache Pollution Detection Method Based on GBDT in Information-Centric Network. *Security and Communication Networks*, 2021(1), 2021.

[86] Gordon Meiser, Pierre Laperdrix, and Ben Stock. Careful Who You Trust: Studying the Pitfalls of Cross-Origin Communication. In *ASIACCS 2021 - 16th ACM Asia Conference on Computer and Communications Security*, 16th ACM Asia Conference on Computer and Communications Security, Hong Kong / Virtual, China, June 2021.

[87] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. Cached and Confused: Web Cache Deception in the Wild. In *USENIX Security Symposium*, 2020.

[88] Seyed Ali Mirheidari, Matteo Golinelli, Kaan Onarlioglu, Engin Kirda, and Bruno Crispo. Web cache deception escalates! In *31st USENIX Security Symposium*

(USENIX Security 22), pages 179–196, Boston, MA, August 2022. USENIX Association.

[89] Philipp Müller, Niklas Niere, Felix Lange, and Juraj Somorovsky. Turning attacks into advantages: Evading http censorship with http request smuggling. *Proceedings on Privacy Enhancing Technologies*, 2024.

[90] Jens Müller. On Web-Security and -Insecurity: CORS misconfigurations on a large scale, July 2017.

[91] NGINX. NGINX Content Caching. `https://docs.nginx.com/nginx/admin-guide/content-cache/content-caching/`.

[92] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack. In *ACM Conference on Computer and Communications Security*, 2019.

[93] Mark Nottingham. The Cache-Status HTTP Response Header Field. RFC 9211, 2022. `https://datatracker.ietf.org/doc/html/rfc9211`.

[94] OWASP. Cross Site Scripting Prevention Cheat Sheet.

[95] OWASP. Types of XSS.

[96] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 653–665, New York, NY, USA, 2016. Association for Computing Machinery.

[97] Hyundo Park, Indra Widjaja, and Heejo Lee. Detection of Cache Pollution Attacks Using Randomness Checks. In *IEEE International Conference on Communications*, 2012.

[98] Kailas Patil and B. Frederik. A measurement study of the content security policy on real-world applications. *International Journal of Network Security*, 18:383–392, 03 2016.

[99] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, 2019.

[100] Portswigger. Http request smuggling.

[101] Apache HTTP Server Project. Apache Module mod_cache – CacheHeader Directive. `https://httpd.apache.org/docs/2.4/mod/mod_cache.html#cacheheader`.

[102] Judith Ramsay, Alessandro Barbesi, and Jenny Preece. A psychological investigation of long retrieval times on the world wide web. *Interacting with Computers*, 10(1):77–86, 1998. HCI and Information Retrieval.

[103] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex security policy? a longitudinal analysis of deployed content security policies. In *Proceedings of the 27th Network and Distributed System Security Symposium*, 2020.

[104] Sebastian Roth, Lea Gröber, Michael Backes, Katharina Krombholz, and Ben Stock. 12 angry developers - a qualitative study on developers' struggles with csp. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 3085–3103, New York, NY, USA, 2021. Association for Computing Machinery.

[105] Alex Rousskov and Duane Wessels. High-performance Benchmarking with Web Polygraph. *Software: Practice and Experience*, 34(2):187–211, 2004.

[106] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[107] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Clock around the clock: Time-based device fingerprinting. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1502–1514, New York, NY, USA, 2018. Association for Computing Machinery.

[108] Werner Schindler. A timing attack against rsa with the chinese remainder theorem. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, pages 109–124, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[109] Werner Schindler. Optimized timing attacks against public key cryptosystems. *Statistics & Risk Modeling*, 20(1-4):191–210, 2002.

[110] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *Financial Cryptography and Data Security: 21st International Conference, FC 2017, Sliema, Malta, April 3–7, 2017, Revised Selected Papers*, page 247–267, Berlin, Heidelberg, 2023. Springer-Verlag.

[111] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. Same-Origin policy: Evaluation in modern browsers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 713–727, Vancouver, BC, August 2017. USENIX Association.

[112] Andrew Sears, Julie A. Jacko, and Michael S. Borella. Internet delay effects: how users perceive quality, organization, and ease of use of information. In *CHI '97 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '97, page 353–354, New York, NY, USA, 1997. Association for Computing Machinery.

[113] Paula R. Selvidge, Barbara Chaparro, and Gregory T. Bender. The world wide wait: Effects of delays on user performance. In *Ergonomics for the new millennium*, page 416 – 419, 2000.

[114] Bojan Simic. The performance of web applications: Customers are won or lost in one second. *AR Library*, 2008.

[115] Dave Smart and Jamie Indigo. Page Weight. The Web Almanac, 2024. `https://almanac.httparchive.org/en/2024/page-weight`.

[116] Michael Smith, Craig Disselkoen, Shravan Narayan, Fraser Brown, and Deian Stefan. Browser history re:visited. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, August 2018. USENIX Association.

[117] Dolière Francis Some, Nataliia Bielova, and Tamara Rezk. On the content security policy violations due to the same-origin policy. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 877–886, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[118] Squid. Squid: Optimising Web Delivery. `http://www.squid-cache.org/`.

[119] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World*

Wide Web, WWW '10, page 921–930, New York, NY, USA, 2010. Association for Computing Machinery.

[120] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. Who's hosting the block party? studying third-party blockage of csp and sri. In *Network and Distributed System Security Symposium*, 2021.

[121] Ben Stock, Giancarlo Pellegrino, Frank Li, Michael Backes, and Christian Rossow. Didn't You Hear Me? — Towards More Successful Web Vulnerability Notifications. In *The Network and Distributed System Security Symposium*, 2018.

[122] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification. In *USENIX Security Symposium*, 2016.

[123] David Strom. What is Magecart? How this hacker group steals payment card data. CSO Online, 2019. `https://www.csoonline.com/article/3400381/what-is-magecart-how-this-hacker-group-steals-payment-card-data.html`.

[124] Avinash Sudhodanan, Roberto Carbone, Luca Compagna, Nicolas Dolgin, Alessandro Armando, and Umberto Morelli. Large-Scale Analysis & Detection of Authentication Cross-Site Request Forgeries. In *IEEE European Symposium on Security and Privacy*, 2017.

[125] Martin Thomson and Cory Benfield. HTTP/2. RFC 9113, June 2022.

[126] Tim Tomes. Method interchange: The forgotten vulnerability, May 2015.

[127] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. Data exfiltration in the face of csp. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 853–864, New York, NY, USA, 2016. Association for Computing Machinery.

[128] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1382–1393, New York, NY, USA, 2015. Association for Computing Machinery.

[129] Vik Vanderlinden, Tom Van Goethem, and Mathy Vanhoef. Time will tell: Exploiting timing leaks using http response headers. In Gene Tsudik, Mauro Conti,

Kaitai Liang, and Georgios Smaragdakis, editors, *Computer Security – ESORICS 2023*, pages 3–22, Cham, 2024. Springer Nature Switzerland.

[130] Vik Vanderlinden, Wouter Joosen, and Mathy Vanhoef. Can You Tell Me the Time? Security Implications of the Server-Timing Header. In *Proceedings of MADWeb 2023–Workshop on Measurements, Attacks, and Defenses for the Web*. Internet Society, 2023.

[131] Varnish. Varnish HTTP Cache. `https://varnish-cache.org/`.

[132] Varnish. Hashing, 2025. `https://varnish-cache.org/docs/7.7/users-guide/vcl-hashing.html`.

[133] Daniel Veditz, Adam Barth, and Mike West. Content security policy level 2. W3C recommendation, W3C, December 2016. https://www.w3.org/TR/2016/REC-CSP2-20161215/.

[134] W3Techs. HTTP/3 vs. HTTP/2 usage statistics, July 2025, 2025.

[135] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1376–1387, New York, NY, USA, 2016. Association for Computing Machinery.

[136] Michael Weissbacher, Tobias Lauinger, and William Robertson. Why is csp failing? trends and challenges in csp adoption. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, pages 212–233, Cham, 2014. Springer International Publishing.

[137] Mike West and Antonio Sartori. Content security policy level 3. W3C working draft, W3C, February 2023. https://www.w3.org/TR/CSP3/.

[138] WHATWG. Fetch Standard.

[139] Bastiaan Wissingh, Christopher A. Wood, Alex Afanasyev, Lixia Zhang, David R. Oran, and Christian Tschudin. Information-Centric Networking (ICN): Content-Centric Networking (CCNx) and Named Data Networking (NDN) Terminology. RFC 8793, 2020. `https://datatracker.ietf.org/doc/html/rfc8793`.

[140] World Wide Web Consortium (W3C). Cool URIs don't change, 1998. `https://www.w3.org/Provider/Style/URI.html`.

[141] Mengjun Xie, Indra Widjaja, and Haining Wang. Enhancing Cache Robustness for Content-Centric Networking. In *IEEE International Conference on Computer Communications*, 2012.

[142] xxHash. xxHash. `https://xxhash.com/`.

[143] Lin Yao, Zhenzhen Fan, Jing Deng, Xin Fan, and Guowei Wu. Detection and Defense of Cache Pollution Attacks Using Clustering in Named Data Networks. *IEEE Transactions on Dependable and Secure Computing*, 17(6):1310–1321, 2020.

[144] Lin Yao, Yujie Zeng, Xin Wang, Ailun Chen, and Guowei Wu. Detection and Defense of Cache Pollution Based on Popularity Prediction in Named Data Networking. *IEEE Transactions on Dependable and Secure Computing*, 18(6):2848–2860, 2021.

[145] Wang Zhiheng and Phan Doantam. Using page speed in mobile search ranking. `https://developers.google.com/search/blog/2018/01/using-page-speed-in-mobile-search`, 2018.

Appendix A

Path Confusion Techniques

Here, we present a summary of the path confusion techniques we use in our experiments to craft the attack URLs using our novel Web Cache Deception detection methodology. The methodology and the experiments are thoroughly described in Chapter 4.

Table A.1 presents examples for each path confusion technique we use when crafting the attack URLs in our comparative evaluation, and a breakdown of the findings for each. Table A.2 shows a similar summary for the large-scale experiment.

Path Parameter refers to the original WCD technique introduced by Omer Gil, and the remaining 4 encoding techniques listed in the first group of rows were presented by Mirheidari et al. in their paper "Cached and Confused". The second group contains 7 additional path confusion techniques we propose here. While there are overlaps between the websites each technique can exploit, combining all 12 greatly increases the chances of exposing WCD vulnerabilities.

Table A.1: The number of vulnerable websites detected via each path confusion variation over 404 targets in our comparative experiment. The middle rule separates the previously known variations above from the new ones we introduce in this research below. Percentages are calculated over the total number of true positives for each methodology.

Path Confusion Technique	Payload	CC	DE_{auth}	DE
Path Parameter	/random.css	13 (72.22%)	63 (54.78%)	62 (59.62%)
Encoded Newline	%0Arandom.css	7 (38.89%)	90 (78.26%)	90 (86.54%)
Encoded Question Mark	%3Fname=valrandom.css	8 (44.44%)	89 (77.39%)	87 (83.65%)
Encoded Semicolon	%3Brandom.css	9 (50.00%)	90 (78.26%)	90 (86.54%)
Encoded Sharp	%23random.css	9 (50.00%)	89 (77.39%)	88 (84.62%)
Encoded Slash	%2Frandom.css	8 (44.44%)	94 (81.74%)	96 (92.31%)
Double Encoded Newline	%25%30%41random.css	7 (38.89%)	90 (78.26%)	87 (83.65%)
Double Encoded Null	%25%30%30random.css	6 (33.33%)	87 (75.65%)	85 (81.73%)
Double Encoded Question Mark	%25%33%46random.css	8 (44.44%)	90 (78.26%)	86 (82.69%)
Double Encoded Semicolon	%25%33%42random.css	9 (50.00%)	89 (77.39%)	84 (80.77%)
Double Encoded Sharp	%25%32%33random.css	8 (44.44%)	89 (77.39%)	86 (82.69%)
Double Encoded Slash	%25%32%46random.css	7 (38.89%)	84 (73.04%)	88 (84.62%)

Table A.2: The number of vulnerable websites detected via each path confusion variation in the large-scale measurement over the Alexa Top 10K. The middle rule separates the previously known variations above from the new ones we introduce in this research below. Percentages are calculated over the total number of findings.

Path Confusion Technique	Payload	DE
Path Parameter	/random.css	618 (52.02%)
Encoded Newline	%0Arandom.css	528 (44.44%)
Encoded Question Mark	%3Fname=valrandom.css	801 (67.42%)
Encoded Semicolon	%3Brandom.css	863 (72.64%)
Encoded Sharp	%23random.css	526 (44.28%)
Encoded Slash	%2Frandom.css	559 (47.05%)
Double Encoded Newline	%25%30%41random.css	383 (32.24%)
Double Encoded Null	%25%30%30random.css	349 (29.38%)
Double Encoded Question Mark	%25%33%46random.css	387 (32.58%)
Double Encoded Semicolon	%25%33%42random.css	402 (33.84%)
Double Encoded Sharp	%25%32%33random.css	386 (32.49%)
Double Encoded Slash	%25%32%46random.css	365 (30.72%)

Appendix B

Sample GitHub Repositories

Table B.1 lists a sample of the projects we detected on GitHub and analysed for cache configurations. We present the top 10 projects for each caching proxy, sorted by popularity.

Table B.1: Samples of the 8 most popular GitHub repositories where we identified a cache configuration for the 5 selected cache technologies.

Cache	Repository Name	Stars	Size (MB)
ATS	apache/trafficserver	1860	256
ATS	mustafaramadhan/kloxo	336	256
ATS	sqawasmi/trafficserver-docker	19	256
ATS	Li4n0/My-CTF-Challenges	16	256
ATS	sunnyszy/lrb-prototype	9	1048576
ATS	ShufanWangBGM/Reinforcement-[...]	8	524288
ATS	xyp-root/geektime-hands-on-[...]	3	1024
ATS	JasonGiedymin/ats-docker	2	256
HAProxy	haproxy/haproxy	5508	200.0
HAProxy	haproxytech/dataplaneapi	345	1024.0
HAProxy	haproxytech/client-native	133	4.0
HAProxy	http-tests/cache-tests	122	4.0
HAProxy	haproxytech/config-parser	82	4.0
HAProxy	intel/workload-services-framework	55	128.0
HAProxy	existentialcomics/kungFuChess	31	200.0
HAProxy	xen0bit/muvr.xyz	22	64.0
NGINX	Gallopsled/pwntools	12518	1024.0
NGINX	ShaneIsrael/fireshare	757	500.0
NGINX	huasenjio/huasenjio-compose	560	51200.0
NGINX	aldor007/mort	513	73.0
NGINX	PUGX/badge-poser	477	500.0
NGINX	brunobritodev/JPProject.IdentityServer4.SSO	455	1024.0
NGINX	dyc3/opentogethertube	427	1000.0
NGINX	DanWahlin/Angular-Docker-Microservices	220	3000.0
Squid	bannedbook/fanqiang	39502	5000.0
Squid	vimagick/dockerfiles	3181	100.0
Squid	av/harbor	1576	100.0
Squid	xjdrew/kone	705	10240.0
Squid	diladele/squid-windows	194	100.0
Squid	salrashid123/squid_proxy	128	100.0
Squid	1265578519/PAC	54	5000.0
Squid	jacobproject/operation	40	800.0
Varnish	varnish/Varnish-Book	353	256.0
Varnish	wenerme/wener	300	32768.0
Varnish	aliuosio/mage2.docker	58	2048.0
Varnish	aqzt/docker-alpine	53	100.0
Varnish	localwiki/localwiki-backend-server	48	712.0
Varnish	camptocamp/puppet-varnish	36	256.0
Varnish	roadiz/skeleton	11	256.0
Varnish	silentred/learning-path	8	256.0