Encrypted Endpoints: Defending Online Services from Illegitimate Bot Automation

August See
Universität Hamburg
Hamburg, Germany
richard.august.see@uni-hamburg.de

Kevin Röbert
Universität Hamburg
Hamburg, Germany
kevin.roebert@uni-hamburg.de

Mathias Fischer
Universität Hamburg
Hamburg, Germany
mathias.fischer@uni-hamburg.de

ABSTRACT

Automated usage of web services by programs, known as bots, poses risks such as data scraping, spam, and cyber attacks. For instance, X suffers from millions of bot accounts typically controlled by relatively fewer adversarial organizations to create fake likes and comments. The most widely used solution to distinguish humans from bots (CAPTCHA) is perishing due to advances in machine learning. Obfuscation techniques in binaries, applications, or websites are designed to impede the creation of bots but fail to prevent their scalability. Bypassing these measures often requires only a one-time effort. We propose *encrypted endpoints* as a novel strategy to combat the scalability of web bots, particularly in scenarios where bots leverage multiple accounts. For that we assign unique endpoints (URLs) to each user account, thereby restricting bot applicability across different accounts and necessitating the extraction of account-specific endpoints per bot instance. Our approach is applicable to a wide range of services utilizing endpoints, including desktop and mobile applications, web applications, and even static or HTML-only websites. We implemented our approach directly within a backend framework and observed that the latency overhead is less than 0.1ms per request, which constitutes less than 1% of the total request time. Our solution, developed as simple middleware, can be easily integrated in existing projects with low effort. Additionally, we have extended our approach to the Jinja2 template engine, thereby supporting encrypted endpoints for websites out of the box. Our analysis indicates that our approach not only effectively protects against simple bots but also, when coupled with obfuscation techniques, further impedes bot creation.

CCS CONCEPTS

• **Security and privacy → Web application security**; *Software reverse engineering*.

KEYWORDS

web bots, obfuscation, endpoints

ACM Reference Format:
August See, Kevin Röbert, and Mathias Fischer. 2024. Encrypted Endpoints: Defending Online Services from Illegitimate Bot Automation. In *The 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024), September 30–October 02, 2024, Padua, Italy*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3678890.3678918

1 INTRODUCTION

The automated use of Internet services is an integral part of the Internet and the Web, especially as web services increasingly depend on each other. This ranges from retrieving resources from CDNs to price comparison portals to unauthorized automation, e.g., by bots in social media. In the following, the unauthorized automated use of a service by scripts or computer programs is called a *bot*. In addition, we focus on bots that gain an advantage by creating many accounts.

While automation is a drive of our Internet, some services should not be automated, as this can lead to financial and even social damage. Bots significantly increase the load of services (bad bots 27.7% and automated traffic in total 42.3% in 2021 [2]) and thus increase infrastructure costs. This applies to all services, but there are certain services where bots can do greater harm. Social media, for example, is intended for human users only. Bots that automatically create or control multiple accounts can be used on a large scale to spread false information and opinions. This does not only increase infrastructure costs but also affects the satisfaction of human users. Beyond that, bots have been successfully used in the past to influence elections [9].

Utilizing the Application Programming Interface (API) or endpoints of a service is the most efficient method for bot development, known as API/endpoint-based bots. These bots, which operate through HTTP requests from scripts or automated browsers, can be easily created using automated tools [24, 34]. These tools facilitate extracting service interaction data, simplifying session replays. Given their prevalence, our paper focuses on defenses against these bots, as discussed in Section 2.

The main problem of bots is something all bot types have in common. Once written, they are easy to duplicate and thus to scale. For example, a X bot that automatically likes everything with a specific tag, e.g., *#conference* will work for every account. This ability to scale is what makes bots so dangerous. While one bot likely will not have much negative effect, an army of bots will. Using the endpoint directly is most natural and scales better than automation via the user interface, so we focus on how such bots can be restricted.

More and more companies are using anti-bot solutions, such as CAPTCHA or obfuscation approaches.CAPTCHAs are the most popular defense against bots. They use problems that are easy for humans to solve but difficult for computers. However, the number of problems that fall into this category is decreasing with the advancement of machine learning [7, 35]. Additionally, CAPTCHAs negatively impact the user experience and waste time, even when modern risk assessment approaches do not prompt every user with a CAPTCHA challenge [1]. As a result, obfuscation approaches are used to hinder the creation of bots by making it more difficult for bots to extract information [40], such as URLs, email addresses, or information about the availability of specific products, e.g., graphic cards. However, many bots do not need to extract information, e.g., those that replay sessions or that use the API directly. For example, for spamming comments a bot needs to know the endpoint location (URL) and what data is accepted. Existing obfuscation approaches leave this aspect out because it changes the way the web works. While, our proposed obfuscation approach addresses this gap.

Our main contribution are unique encrypted endpoints so that bots cannot be scaled easily anymore. In more detail, we make the following contributions:

- We introduce encrypted endpoints to hinder the scaling of bots. This is achieved by assigning unique endpoints to each account in a service. Thus, every bot is only valid and usable for a specific account. We call this encrypted endpoints. Existing obfuscation techniques for Binaries [12] or for HTML and JavaScript [14, 40, 44], obfuscate the "application" itself but not the endpoints. Thus, they offer no protection against bots that use the endpoints directly, e.g., using python requests or which replay previously recorded sessions [24, 34]. Our approach and code obfuscation work in tandem, complementing each other. While our approach safeguards against session replays and API bots, code obfuscation adds an extra layer of complexity, rendering the extraction of client-specific encrypted endpoints more challenging. By altering the endpoints for each client periodically, the effort required by an attacker can be further increased. It is worth noting that while our approach was initially designed for accounts, it is also applicable to services without a login feature.
- We purpose methods to enhance the usability of our approach for both users and service providers. So, despite unique URLs, those URLs can still be shared between users without impacting the efficacy against bots. Further, service providers do not have to predetermine all of their endpoints to utilize our approach.
- We implement the approach as a middleware for FastAPI and the Jinja2 templating engine. The source can be found here[1] as well as simple demonstration in form of a video.
- We evaluate our approach's performance and organizational overhead and discuss the additional effort required for attackers to scale bots.
- As an added benefit, our approach can protect against directory traversal attacks and attacks that rely on guessing or injecting data into the path or parameters, as our method renders URLs non-guessable and user-specific.

Note that the primary objective is to increase the effort to scale bots by preventing a bot to be used on a different account, and not the creation itself. While our approach alone achieves this, it **heavily** benefits from being used together code obfuscation to make the extraction of client-specific encrypted endpoints more challenging (cf. Section 5.1). Legitimate bots, security testing, and interoperability across different services are still possible, e.g., by providing special API keys after thorough verification.

The remainder of this paper is structured as follows. Section 2 introduces our threat model, specifying the types of bots we aim to protect against, and describes the attacker model used throughout this paper. Section 3 outlines the requirements for bot defense and reviews related work in the field. Section 4 describes our approach to encrypted endpoints and discusses potential optimizations to restore functionalities such as link sharing. Section 5 presents our evaluation of the proposed approach and discusses its effectiveness in defending against the specified attacker model. It also explores the limitations of our approach. Section 6 doutlines the requirements for code obfuscation techniques that can be integrated with our method. Finally, Section 7 concludes the paper.

2 THREAT AND ATTACKER MODEL

In our threat model a service is accessed by bots. The service either cannot or chooses not to depend on more robust user authentication methods, such as phone verification or presenting personal identification documents. This decision is grounded in realism, as authentication processes that introduce friction tend to deter users from engaging with the service [18]. The service has already implemented account-bound rate limiting, ensuring that actions like purchases and upvotes are constrained within certain limits per account. This setup necessitates a logical reason for bot creators to create multiple bots, each controlling distinct accounts. This scenario is common across various types of websites, including those in social media, e-commerce, and gaming. However, for websites that solely provide information, our approach is not applicable.

For example, on most social media platform, content visibility is influenced by upvotes. Each account is permitted to vote only once, making it advantageous for bot operators to control multiple accounts for the purpose of artificially boosting the visibility of specific content through coordinated upvoting.

2.1 Considered Bots

Our defense mechanisms against bots, specifically targets API-based bots. Our rationale is that API-based bots are prevalently used, easily scalable, and consequently represent a substantial threat. Delving further into the matter:

API-based bots are simpler to develop and maintain because they interact directly with a service's endpoints, circumventing the complexities associated with graphical user interfaces (GUIs). In contrast to GUI elements that are prone to frequent changes, API endpoints, particularly those that are versioned (for example, "/v1/user"), provide a stable interface for automation. This stability diminishes the necessity for continual updates in contrast to bots operating on the UI. However, with current technological advancements, UI-based bots are expected to become more adaptable shortly.

Furthermore, API-based bots have the advantage of automated tools that facilitate the creation of bots by extracting endpoints and data from service interactions. These tools enable straightforward session replays [24, 34], making API/endpoint-based bots a more viable option for our research, which emphasizes efficient and stable automated interactions.

[1] https://github.com/8mas/encrypted-endpoints

API-based bots also require significantly less computational resources compared to their UI-based counterparts, which need to render graphical interfaces and are tightly bound to the underlying program and UI. For each instance of a UI-based bot, a separate interface instance is required, leading to escalated resource consumption and higher scaling costs. This contrast is especially pronounced in contexts like gaming and mobile applications, where UI-based bots are resource-intensive, and scaling becomes more costly.

2.2 Attacker Models

We define two attacker profiles:

Endpoints Only (EO) Attacker. : This EO attacker strategy focuses solely on utilizing service endpoints to develop bots. By recording network traffic, attackers can use tools such as *mitmproxy2swagger* and *charles-extractor* [24, 34] to extract endpoints and data formats from the session, facilitating session replay and bot creation with minimal technical expertise required. This approach bypasses the need to parse desktop binaries, mobile applications (APKs, IPAs), or web elements (HTML, CSS, JavaScript), offering a streamlined method to create efficient bots.

Endpoints and Data Parsing (EP) Attacker. : Contrary to the first attacker model, this EP attacker involves utilizing service endpoints and processing and parsing available data. In the context of mobile and desktop programs, this includes binaries and apps (APKs, IPAs), which are inherently difficult to *parse* and often require reverse-engineering using disassembly and debugging tools. In the context of the web, this encompasses server responses in formats such as HTML, XML, and JavaScript. This method is typically adopted for services that lack a comprehensive API. Direct interaction with specific endpoints to obtain structured data, such as JSON (e.g., "/v1/product/id" providing product prices and availability), is generally more straightforward and reliable than parsing semi-structured data like HTML. The latter, often relying on tools such as XPath or CSS selectors, is more susceptible to errors and complications due to potential changes in the data's structure or format [26].

3 REQUIREMENTS AND RELATED WORK

3.1 Requirements for Bot Defense Approaches

We have identified several requirements for bot defense approaches:

- *Transparent to user*: The approach should not disrupt the user experience, thus avoiding time wastage through mechanisms such as captchas [1] and preventing loss of revenue due to additional hurdles in website navigation [18].
- *Zero data collection*: No data should be collected to differentiate between humans and bots, thereby ensuring privacy. This includes overt measures like requiring an ID to use a service, as well as subtler techniques that collect data such as IP addresses and mouse movements [27].
- *Seamlessly integrable*: The approach should be seamlessly integrable into existing services and codebases with minimal effort.
- *Small performance overhead*: The approach should have minimal impact on the performance of the service, ensuring it remains usable.
- *UI and domain agnostic*: The approach should be UI-agnostic to allow versatile application across different user interfaces. Additionally, it should be domain-agnostic, suitable for use in various contexts such as desktop applications, mobile apps, or HTML-only websites.
- *Resistance against attackers (EO, EP)*: The approach should effectively counter the described attacker models (EO and EP).
- *Open source*: Ideally, the approach should be open source to ensure developers can adapt it as needed.

3.2 Related Work

We divide the related work into three categories. Proving Humanity, Anomaly Detection, and Anti-Analysis. The approach we take in this paper falls into the Anti-Analysis category. A high-level overview of how our approach compares to related work is given in Table 1.

Requirements	CAPTCHA [27, 28]	Brewer et al. [11]	See et al. [32]	Code obfuscation. [12, 36, 39]	User specific API keys	Facebook Anti-Tracking[3]	Our System
Transparent to users		●	●	●	●	●	●
Zero data collection		●	●	●	●	●	●
Seamlessly integrable	●	●		◐	◐		●
Small performance overhead	●	●	◐	◐	●	●	●
UI and domain agnostic	◐	◐	◐	◐	◐		●
Resistance against attacker (EA)	●		●		●		●
Resistance against attacker (EDA)	●		◐[2]		◐[2]		◐[2]
Open source		●	●	◐[3]	◐[4]		●

◐ partially fulfilled ● fulfilled

Table 1: Related bot defending techniques.

Proving Humanity. Approaches in this category aim to prove a user's humanity directly, for example, through possession or human knowledge. The strongest proof that a user is a human would be requiring an official government-issued ID. While this would certainly make it hard to scale bots, it is also a privacy-unfriendly requirement. Even if users did not care about privacy, it would disrupt the user experience, directly impacting companies' revenues [18]. A very well-known, and at present, the standard way to prove humanity is CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) [27, 28]. CAPTCHA relies on problems that are hard to solve for computers and easy for humans,

[2]Only when coupled with code obfuscation approaches.
[3]Some implementations like Tigress are available.
[4]This is more a generic technique than an implementation.

such as reading distorted characters or selecting specific images. This approach, however, has some serve limitations and is becoming less and less effective. First, CAPTCHA breaks the user experience. Second, the time wasted to prove humanity is 500 years each day, alone for CAPTCHA issued by Cloudflare [1]. To deal with this, there is research to decrease the number of issued challenges to users, e.g., privacy pass [13]. In addition, for each correctly solved CAPTCHA, the users receive some tokens. These tokens do not leak information about the user and can be used to bypass CAPTCHA. However, the main problem of CAPTCHA is that the number of problems that are hard to solve for computers and easy to solve for humans is decreasing because of the advancements in machine learning [19, 25, 35].

To address the problems of CAPTCHA, there are developments to attest humanity through cryptographic routines, Trusted Plattform Modules (TPMs), and Trusted Execution Environments (TEEs). The idea is that by possessing a rare resource, e.g., security keys [43], Iphones [20], etc. CAPTCHA no longer has to be solved. Here it must be ensured that no information about the actual user is leaked, but also that a simple (virtual) transfer of the resource is not possible. These approaches [20, 43] are compatible with the pricacy pass [13] protocol and are designed to avoid displaying CAPTCHA altogether.

Anomaly Detection. These approaches try to detect bots through their characteristics, e.g., user-agent or display size and behavior. CAPTCHA providers use these approaches for an initial risk assessment to decrease the number of CAPTCHA presented to users [27, 28]. A harder or no CAPTCHA is presented depending on the calculated score. However, characteristics such as IPs, user agent, resolution, and cookies are needed to calculate the risk. This information can fingerprint the system and track users across multiple websites. In addition, mouse movements and keystrokes allow the user to be identified, not just the browser or system [5, 33]. This is a major criticism of current CAPTCHA systems.

Brewer et al. [11] introduce another way that resembles the idea of honeypots. Every link on a web page is surrounded by fake links that are not visible to the user. A bot that tries to crawl or scrape the website will likely visit a fake link and is thus exposed. In contrast to other anomaly detection approaches, this is very privacy-friendly and stands out positively due to low false positives.

Last, anomaly detection approaches for bot detection suffer from one common problem. If a bot behaves exactly like a human user (characteristics and behavior), distinguishing between humans and bots is no longer possible. However, forcing a bot to behave like a human is a great achievement since it increases the cost of creating undetected bots. Bot writers then need to consider the characteristics and behaviors of humans on a website, and the written bots are not as effective as they could be. However, these advanced bots will not be detectable through anomaly detection alone.

Anti-Analysis. These approaches aim to complicate the creation of bots and increase the costs associated with extracting information, thereby aligning with the objectives of this paper. We describe the requirements for code obfuscation used in conjunction with our approach in Section 6.

The primary challenge in anti-analysis engineering lies in quantifying the efficacy of a technique against an undefined attacker [6, 8, 38]. When an attacker fully controls a device, all anti-analysis techniques can merely elevate the cost for an attacker to reverse engineer an application without the possibility of completely preventing it.

In the context of web bots, obfuscation techniques serve to complicate the extraction of essential information required for bot creation, such as accepted protocols, endpoints, or API keys. Numerous methods exist both for data extraction from a program and for its obfuscation.

However, more than obfuscating the location of elements is required. For many bots, e.g., spambots, only the endpoint and the data format are needed. Consider Listing 1 website that displays a URL with GET parameters and a form. Even if the location is obfuscated (cf. listing 2), the endpoint (`example.com/api`, `example.com/api/user`) and the parameters (`param1`, `name1`) are always the same. Thus, once known, e.g., through recording the traffic, they can be used indefinitely, and the bot can be scaled easily.

Listing 1: Unobfuscated HTML

```
<a id="link1"
    href="example.com/api?param1=hello">Link1</a>
<form action="example.com/api/user" method="POST">
    <input name="name1" id="input1" value="world"/>
</form>
```

Listing 2: Obfuscated HTML (IDs, XPath)

```
<form action="example.com/api/user" method="POST">
    <input name="name1" id="random2" value="world"/>
</form>
<a id="rand1"
    href="example.com/api?param1=hello">Link1</a>
```

There are many approaches that address code obfuscation in the web context. These include academic papers, free-to-use tools [10, 23, 45], and commercial software [21, 22, 29]. Some commercial solutions offer comprehensive packages that handle all three aspects at once [21, 29]. While most approaches claim minimal overhead, they often do not evaluate it.

It is ideal to have an approach where the resource cost of creating the obfuscation is low, and ideally, each client receives its own obfuscated version of the website. While some approaches are non-deterministic, the majority are deterministic, necessitating an additional randomization step. However, websites can be pre-obfuscated, and the obfuscated versions can then be distributed to clients. Most tools are user-friendly and do not require extra configuration for the frontend. Typically, users need only to give their final HTML, JS, or CSS files as input, which are then obfuscated.

A paper analyzing the top 10K Alexa websites found that less than 0.4% use obfuscation on JavaScript [30] and 68.8% use minimization. Another paper analyzing the top 100K Alexa websites found that 0.67% of scripts are obfuscated, but 38% are minimized. While these percentages are not high, they indicate that code obfuscation is employed on some of the most popular websites.

Vikram et al. [40] address this. They build a tool, NOMAD, to defend against web bots without breaking the website for human users. The tool randomizes the Name and ID parameters of HTML form elements for each session. Thus, forcing the bot to extract the correct names and IDs for each session. However, this is limited to only forms and does not apply to (GET) parameters.

Wang et al. [41] introduce WebRanz, a novel mechanism for circumventing ad-blockers by employing randomization techniques to mutate HTML elements and their attributes without affecting the visual appearance or functionality of web pages. This approach invalidates the pre-defined patterns utilized by ad-blockers, thereby enabling content publishers to deliver advertisements effectively. WebRanz also provides a defense against web bots that manipulate DOM objects using similar pattern-matching techniques. The authors evaluate the system on 221 Alexa top web pages and eight bot scripts, demonstrating that WebRanz successfully evades ad-blockers and mitigates the impact of bot scripts with minimal overhead. It is a promising candidate to use in conjunction with our approach to counter the *Endpoint and Data Parsing Attacker*.

See et al. [32] follow a similar path. Their approach assigns a new application protocol for every user. Thus all protocol messages are unique, and bots cannot be scaled. The main problem of this work is that it is not easily usable for HTTP, and to be lightweight needs much fine-tuning.

The examples provided thus far have predominantly pertained to the web context. However, the same principles apply to binary obfuscation techniques. Examples of sophisticated software protection systems include Tigress [12], VMProtect [36], and Themida [39]. These systems are directly applied to the source code of an application, yielding a protected executable that utilizes a range of anti-reverse engineering and obfuscation techniques, thus complicating the extraction of information such as endpoints from compiled binary. However, despite the sophistication of these systems, they struggle to effectively obfuscate endpoints. This limitation arises because data transmission to these endpoints can be observed, for example, by utilizing a system-wide proxy. Consequently, bots can be scaled with relative ease once again.

User-specific API keys that are located in the client application and CSRF tokens might be used to impede bot creation, but original serve distinct purposes. For example, API keys may be used for identifying and authenticating a user across sessions, providing a persistent form of security, whereas CSRF tokens are designed to protect against cross-site request forgery attacks by ensuring that each request to a server is accompanied by a unique token, verifying the request's legitimacy. These mechanisms can be considered predecessors or foundational elements for our technique aimed at impeding the scaling of bots. Essentially, our method applies the concept of a CSRF token across a broader domain, compelling bot creators to extract it to make server-accepted requests. API keys in applications can offer similar functionalities by requireing an attacker to first reverse engineer the API key to authenticate custom requests. Our design, in contrast, is more versatile than CSRF tokens or API keys. It does not require an execution mechanism (like necessary for API keys to authenticate data), nor is it limited to HTTP. Further, it can operate statelessly. The only necessity is an endpoint identifier, which could include, but is not limited to, URLs. Our methodology and API keys are complementary, not exclusive. Integrating them can further obscure not only the endpoint but also the transmitted data, thus forming a comprehensive defense against direct and indirect attacks.

Facebook is using encrypted URLs to combat URL stripping [3]. Every parameter of a URL is encrypted and signed. Thus, it is no longer possible to drop tracking parameters without invalidating the whole link. As this technique is most similar to our approach, we provide a side by side comparison in Table 2.

Feature	Facebook's Implementation	Our Implementation
Goal	Prevent addons like Clear URL from dropping certain tracking parameters, as they break the link.	Prevent scaling of bots by requiring them to extract URLs from the source or binary.
Method	Encrypt and sign parameters using (likely) a general key, not client-specific.	Encrypt and sign both path and parameters using a client-specific key.
Security	–	Prevents techniques like directory traversing, guessing paths, local file inclusion.
Availability	–	Free and open source.

Table 2: Comparison of URL Protection Strategies

4 ENCRYPTED ENDPOINTS

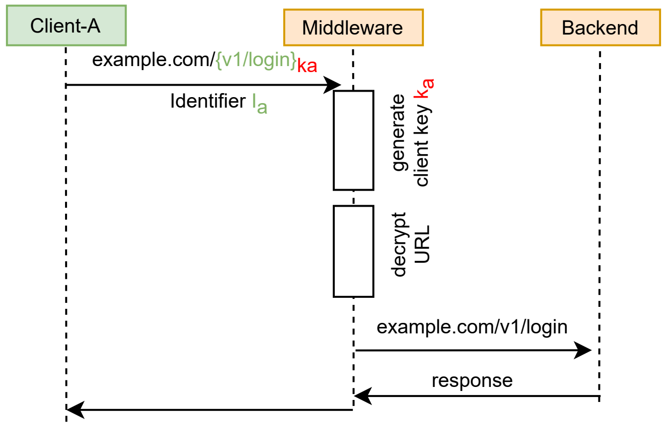
The core idea to limit the scalability of endpoint-based bots is the usage of encrypted endpoints. Every client (user), e.g., distinguished by session cookie or IP, receives a version of the same website with unique URL-Paths and parameters. Paths and parameters are encrypted, signed, and only valid for one client. If a bot is created, it cannot be duplicated easily, as the URLs are only valid for the particular client instance and, i.e., the bot. To create multiple bots, the bot creator is forced to extract the custom URLs from the website. This effectively prevents bots that rely on the simple replay of data [24, 34]. While extraction of URLs is trivial in normal cases, this can be made considerably more difficult by using obfuscation approaches from related work like [11, 14, 40]. Since URL paths and parameters are signed, any modification to them can be detected. A live demonstration of this concept has been implemented in a web application[5]. It is important to note that while the example provided is a web app, for ease of sharing and demonstration purposes, the underlying principle is equally applicable to both computer and mobile applications.

4.1 Basic Approach

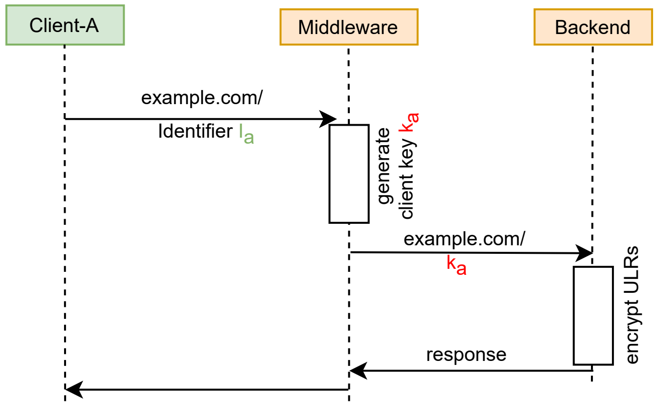
Figure 1 outlines the proposed methodology. The encrypted endpoints middleware, which can be integrated with the server, receives requests from clients. The middleware generates a secure client key from the client identifier (e.g., a user ID) as detailed in Section 4.3.1. We assume that this identifier is non-forgeable.

In a typical scenario as seen in Figure 1a, the client sends a request to an encrypted endpoint, such as a URL, accompanied by its client identifier. The middleware, upon receiving this information, generates a client-specific key to decrypt the URL. If decryption is successful, the URL is forwarded to the backend server, which then retrieves the response and sends it back through the middleware

[5]https://github.com/8mas/encrypted-endpoints



(a) Encrypted Endpoint Usage Backend Only



(b) Encrypted Endpoint Usage in Context of Webpages and HTTP

Figure 1: Overview of Encrypted Endpoint Usages

to the client. It is important to note that clients are incapable of generating valid URLs independently; instead, the middleware and backend collaborate to create and distribute these URLs to the client, as illustrated in Figure 1b. In this process, the initial request from a client is directed to an unencrypted endpoint, for instance, *example.com/*. The middleware subsequently generates the client key and relays the request to the backend. The backend server then retrieves the resources and encrypts all URLs within these resources using client-specific encrypted endpoints.

The encrypted response is then transmitted back to the client, enabling the use of these encrypted endpoints as shown in Figure 1a. Navigation on the client's side proceeds normally, with additional URL requests being validated and decrypted by the middleware. For applications such as desktop and mobile apps, these endpoints can be pre-populated during compilation or installation, enhancing the security and functionality of the system.

Figure 2 illustrates the efficacy of encrypted endpoints in mitigating bot activities and preventing accidental exposure of sensitive files. In Figure 2a, Client-B attempts to access a URL specifically generated by the server for Client-A. As part of this process, Client-B sends its identifier, prompting the middleware to generate the client-specific key k_b and attempt to decrypt the accessed URL. This attempt fails due to a Message Authentication Code (MAC) mismatch, as the URL was encrypted using k_a, resulting in an error response. This mechanism ensures that bots configured for one account cannot be repurposed for another, enhancing security. However, it is important to acknowledge the unintended consequence of inhibiting link sharing, a potentially undesirable outcome. We address this issue in Section 4.3.3.

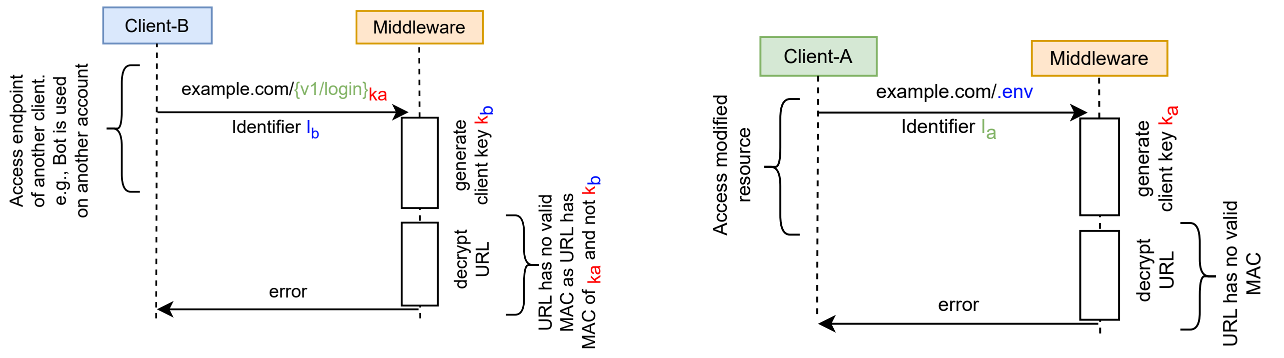
Figure 2b depicts a scenario where a client attempts to access a URL not issued by the server, such as during an attack aimed at discovering sensitive files through techniques like environment variable file scanning or directory traversal. The principle remains consistent with the previous example, where the middleware generates k_a based on the client's provided identifier. Since the accessed URL lacks a valid MAC, an error is returned, and access to the resource is denied. This feature of our approach effectively acts as a URL whitelist, allowing access only to URLs returned by the server to the client. Consequently, this strategy significantly reduces the risk of resource guessing, scanning for sensitive files, and mitigates certain attack vectors such as directory traversal and SQL injection in specific contexts (refer to Section 5.3).

This construction compels bot creators to dynamically search for the current endpoints, representing a significant step forward in impeding bots that rely on hardcoded or memorized endpoints [24, 34]. Nevertheless, extracting endpoints from sources such as HTML, binary code, software, Java, applications, or similar materials is not inherently challenging. Therefore, our approach gains a considerable advantage by incorporating obfuscation techniques that increase the difficulty of identifying specific endpoints.

It is essential to recognize that relying solely on code obfuscation is inadequate for deterring bots. Automated tools, as mentioned in [24, 34], can efficiently extract all necessary endpoints and data from a recorded session, facilitating the easy creation of session replays and bots.

4.2 Formal Model

An application, e.g., a website W contains a list of URLs, $U = \{u_1, ..., u_n\}$. We filter URLs from this list, that either cannot be encrypted, e.g., if the resource is located on a third party location or should not be encrypted, e.g., if it is a public shared URL (cf. Section 4.3.3). While normally URLs contain more, e.g., the scheme, for simplicity, our URLs only contain a path p and a set of parameters a. Any operation on a URL u_i is on the concatenated string of path and parameters, i.e., $u_i = p_i || a_i$. A client c has a unique identifier I_c. The server has a main key k_m. Using a key derivation function $KDF(k_m, I_c)$, the server generates a client key k_c. If identifiers are reused between clients, e.g., IP addresses, a nonce that is regularly changed should also be included in the KDF. The client key encrypts and authenticates each URL $u \in U$ of a website using authenticated encryption (AE) like AES-GCM, which return the encrypted message e as well as a Message-Authentication-Code (MAC) t_e. The URL line must not leak anything about the URL's semantics. Thus,



(a) Invalid URL for Client B because the URL was generated for Client A, rendering the MAC invalid. **(b) Invalid URL due to the client's direct access attempt on a URL not issued by the server.**

Figure 2: Overview of Potential Errors and the Effectiveness of Encrypted Endpoints in Enhancing Security.

a MAC alone is not enough. Since URLs might be differentiable by their length, padding can be used to increase the length of URLs.

$$u' = \langle e, t_e \rangle = AE_{k_c}(u) \tag{1}$$

In this equation, u' is the newly constructed URL that replaces URL u. The encrypted path and parameters are e, and t_e is the message authentication code of e. Both are constructed using the client key k_c, only known by the middleware. The client then receives a version of the web page W' where every URL $u_i \in U$ is replaced with u'_i. In the subsequent figures, we employ the shorthand notation $\{url\}_{key}$ to denote authenticated encryption of URLs.

This construction is stateless, i.e., the middleware does not need to save any client keys as they are derived from the client identifier sent with every request. Note that encrypting URLs is necessary, and a MAC is not sufficient because URLs can be identified by their paths and parameters (cf. Section 4.3.1 and Section 5.1). A stateful construction can save CPU time but uses space as compensation, i.e., each custom URL must be stored for each client.

As long as the client identifier is not easy to share and secure cryptographic primitives are used, these constructions allow URLs to be used only by the respective client. No URLs can be created that the server did not create itself.

4.3 Using Encrypted Endpoints

This section describes how encrypted endpoints could be used productively. The choice of the client identifier, possible error handling, and possible optimizations are discussed.

4.3.1 Choosing the Client Identifier. We operate within a designated threat model, where mechanisms such as rate limiting are implemented (cf. Section 2). The client identifier, depicted in Figure 1 serves as the foundation for generating the corresponding client key, a crucial element utilized in URL encryption. To preempt duplication, the client identifier must be associated with a resource that proves challenging to replicate or that can be subject of rate limiting. For example, one can achieve this by constraining account activities and limiting actions based on User IDs, IP addresses or browser fingerprints. This safeguard is necessary since the URLs generated will be identical for a given client identifier.

There are several ways to choose client identifiers or even to combine different factors. Here, we will discuss the advantages and disadvantages of the identifiers User ID, IP address and browser fingerprint. However, combining different identifiers can improve the robustness of the system.

Session Cookie / User ID In scenarios where the service incorporates encrypted endpoints alongside a login mechanism, a user ID or session cookie is used to identify a client. This setup ensures that URLs are customized for individual accounts, thereby anchoring bots to specific accounts.
Note that although duplicating a session cookie is feasible, this offers no advantage to the attacker. The subsequent bot would be confined to the same account without any incremental capabilities. For example, it could neither cast additional upvotes on the same post on a social media platform nor purchase more products than its predecessor. Despite the potential feasibility of generating valid session cookies, perhaps through automated registration, the differentiation provided by unique user IDs mandates that bots be specifically tailored to their respective accounts.However, is its reliance on a login feature, which might not be applicable for all services. In such cases, alternative identifiers should be considered.

IP Address The IP address constitutes a practical identifier for a broad range of applications, automatically accompanying each request. To prevent IP address spoofing, this identification method relies on the exchange of multiple packets, such as those involved in a full TCP handshake, which helps deter trivial alterations of the source IP. However, the use of IP addresses as client identifiers presents challenges, such as the invalidation of all URLs upon an IP address change—occurring when a user switches from mobile data to Wi-Fi—due to the resultant alteration in the derived client key. Strategies to address these concerns are discussed in Section 4.3.3. It is crucial to enforce a per-IP rate limit to thwart the repeated initiation of a single bot from the same IP. Additionally, the coexistence of multiple devices behind

NAT, sharing an IP address, necessitates the combination of IP and port for a more effective client identifier construction.

Device/Browser Fingerprint Utilizing a device or browser fingerprint generated on the client side serves as an effective method for identifying users accessing the service through an app or browser. This technique surpasses the limitations of easily spoofable attributes, such as browser version and installed fonts, by incorporating at least one distinctive and difficult-to-replicate feature. For instance, leveraging a canvas fingerprint allows for the unique identification of devices based on their graphics processing capabilities [31]. The primary challenge with this method is the reliance on executing client-side code, such as JavaScript, to obtain the fingerprint. This necessity might restrict its applicability in environments where client-side code execution is disabled.

While numerous techniques exist for creating an identifier, the optimal solution is undoubtedly the use of a user/client/account ID, generated after login. This method is inherently unique to each client, thereby circumventing issues associated with IP-based identifiers, which may change when switching networks.

For services not requiring authentication, a combination of the non-forgeable IP address and browser fingerprint is potentially the most effective approach. This is because, even if the IP address changes, it does not necessarily disrupt the user's session.

In the subsequent sections, we will focus on addressing challenges related to URL sharing and session resumption.

4.3.2 Partial Encrypted Endpoints. Our methodology is optimally applied to endpoints that are fully known beforehand by the server. However, this is not always feasible. For instance, search fields that accept arbitrary user input and incorporate it into a parameter present a challenge. In such cases, only the parts of the URL known to the server in advance can be encrypted, while the dynamic, user-controlled portion must be appended. This approach necessitates distinguishing between encrypted and unencrypted parts, which can be achieved, for example, by using delimiters or length fields.

If the application cannot modify itself at runtime, such as HTML without JavaScript, and defense against the EP attacker is required, no static part of the URL should remain unencrypted. This could be a vector for differentiating URLs. Furthermore, URLs should be padded to equal or random lengths so that an attacker cannot differentiate them by length.

For applications that can modify themselves at runtime, and depending on the obfuscation used, this might not be a major problem, as the URL can be split into multiple locations and the parts themselves can be obfuscated as well.

When utilizing partially encrypted endpoints, developers must specify in advance which parts are always encrypted and which may remain unencrypted. Failure to do so could enable an attacker to supply unencrypted endpoints and attempt to deduce and use them.

4.3.3 Shared URLs and Session Resumption. Two issues need to be addressed for practical use: URL sharing and session resumption. URL sharing is a widespread practice on the Internet for sharing resources. An online store, for example, relies on customers being able to send product links to each other. Further, session resumption, e.g., switching between devices or networks, is crucial for a smooth user experience.

URL Sharing. With the current design, URL sharing between users is only possible if they have the same client identifier, which should be avoided. The same is true for shared URLs between services, e.g., a online shop that is not using our approach linking to a payment provider that uses encrypted URLs. If the user is already logged in, the link will break.

The service using encrypted URLs could retain certain URLs or operations (e.g., HTTP methods like POST, GET, DELETE) to be used without our approach, i.e., as normal URLs. This could apply to URLs or URL paths that need to be public, such as incoming links that are used by unknown third parties, e.g., the payment site of a payment provider. The drawback of this method is that bots could operate on those URLs normally, so it should be used only when necessary. Note that after accessing the service using the normal URL, all subsequent URLs are encrypted again. Thus, our approach is only diminished for one URL operation. This requires the service to identify URLs and operations that should be public and to make them known to the middleware.

Another approach is to allow URLs to be accessed by a different user while restricting the access to read-only, i.e., only sharing URLs that do not alter state by restricting the operations on the URLs. This requires identifying URLs and operations that do not modify state. If HTTP is used, this should be straightforward, as the method has properties like *safe*, which indicate that they should not alter state. This is defined in the HTTP RFCs[16, 17]; however, it is currently treated as a convention and may be violated by developers. Consequently, safe methods like GET cannot always be relied upon to be safe. This necessitates operators verifying whether their application adheres to the standard. If not, the first method must be used. Thus this approach as well requires the service to identify URLs and operations that should be public and to make them known to the middleware.

This makes URL sharing between users possible, it does not affect the defense against bots. Multiple bots can now access the same URL without requiring the same client identifier. However, this applies only to „read" access. Whether one or more bots have read access to the same resource is mainly inconsequential and offers no additional value for the bot creator. In edge cases, such as displaying the availability of a product, rate limiting could be circumvented. In contrast, multiple bots can benefit the bot creator for operations that alter the state, such as sending spam or purchasing products.

For sharing encrypted URLs, the approaches the middleware needs to be able to process the request, thus the URL needs to be decrypted. In a stateless construction, the client key can be appended to the URL, encrypted, and authenticated with the server key. Thus for a read-only operation, the URL can be decrypted by the middleware by decrypting the client key first.

$$\begin{gathered} u' = \langle e, t_e, g, t_g \rangle \\ \langle g, t_g \rangle = AE_{k_m}(k_c) \end{gathered} \tag{2}$$

New to Equation 1 are the encrypted client key g and the accompanying MAC tag t_g. Additionally, the authenticated encrypted client key k_c is constructed using the secret key k_m, which is known only to the middleware.

In deployment, shared URLs can be identified by adding an identifier to it, e.g., between the client encrypted URL and the client key $\langle e, t_e, \textit{delimiter}, g, t_g \rangle$. It would check if it is a state-modifying request and g is not forged. Then it can decrypt g to obtain k_c. Next, it needs to verify e, t_g and can obtain the URL. In theory, a delimiter is not necessary if g, t_g are of fixed size, but that would force the backend to treat every URL first as a shared URL.

Session Resumption. This issue arises only when an identifier that is prone to change is selected, such as relying solely on the IP address. For instance, a user's IP can change when transitioning from a mobile network to a WiFi network, leading to a scenario where all current URLs become invalid. This is not conducive to a user-friendly experience.

In cases where utilizing a more stable client identifier is not feasible, one straightforward solution is to enable URL sharing. This allows users to access shareable URLs even if their identifier changes.

A stateful approach to addressing this challenge involves the use of one-time tokens, which can circumvent URL verification once. These tokens could be stored as cookies in HTTP contexts. However, it is crucial that the token is valid only for the respective user, meaning it should bypass URL verification exclusively for that user. This can be achieved by storing the client key in an encrypted and authenticated form using the server key, as outlined in a manner similar to Equation 2.

When a client initiates a request from a new IP, resulting in a changed derived client key, the token comes into play. The process begins with the verification and decryption of the token, followed by the verification and decryption of the requested URL. It is important to note that this token is inapplicable to URLs belonging to a different client, as they are authenticated with a distinct client key. Successful access to the new URL leads to the construction of all subsequent URLs with the new client key.

Employing this method for session resumption does not undermine defenses against bots, as it does not offer them any advantage. While a bot may transfer the URL to another bot, accessing the resource is an action the original bot could have executed, maintaining the integrity of the system's bot defense mechanisms.

4.4 Optimisations

4.4.1 Validity period for URLs. In the current design, URLs for a client remain the same if the client identifier does not change. This means that when a client-specific bot is created, it does not have to re-extract the URLs for that client. The client key can be made expirable to increase the effort required to maintain bots. In a stateless design, client keys k_c can be regenerated at specified intervals t, or for each application update, by incorporating a nonce into the derivation of the client key, which would always change, $\textit{KDF}(M, I_c, \textit{nonce}_t)$. In a distributed infrastructure, e.g., using anycast or a load balancer, to avoid syncing the nonce, one could derive the nonce using a deterministic pseudorandom function. Due to our method of session resumption (Section 4.3.3), this does not result in any loss of user experience. Furthermore, any previously shared URLs do not expire.

4.4.2 Minimizing the Overhead. Several optimizations can be made to reduce the overhead or increase the cost of maintaining bots.

Space-Time Tradeoff The current design is stateless and regenerates client keys and URLs on the fly, which might result in increased latency due to the frequent execution of CPU-intensive operations. One optimization is to introduce a state which stores client keys or URLs. Then, if a client requests a URL, the backend only needs to check whether the URL is known. No cryptographic operation is required. The URLs can be cached in a database or memory. However, storing all URLs for each client is storage-intensive. Another less extreme time-space tradeoff is to store the client key, so it does not have to be regenerated for each request. This will decrease the CPU-intensive operations and decrease the latency. We will evaluate these methods in Section 5.2.

Risk Assesment A strong resource optimization is the use of risk assessment approaches. The generation of client-specific URLs is computationally more expensive. By using risk assessment approaches [37, 42], this overhead can be reduced for the server and legitimate clients. Simple examples of such risk assessments include identifying high-volume IPs or clients that have been connected to the service for an unusually long time or checking if the client is an automated browser like Selenium or Puppeteer. This way, the URLs can be generated only for clients that are flagged as high-risk, while the rest of the clients will be served directly from the server or a Content Delivery Network (CDN).

5 EVALUATION

In this section, we summarize the results of our evaluation. The focus is to determine the approach's overhead and compare it to similar obfuscation techniques. In detail, we answer the following research questions:

RQ1: To what extent does our approach restrict the scalability of bots, and how does it perform in comparison with alternative bot detection or mitigation strategies?

RQ2: What is the computational and operational overhead associated with employing encrypted endpoints?

RQ3: Against which specific attacks do encrypted endpoints provide protection?

We implement our approach as middleware in FastAPI, ensuring seamless integration by simply incorporating the middleware into the backend. Our implementation extends beyond the backend, offering support for rendering via Jinja2. This enables the straightforward inclusion of webpages, encompassing HTML and JavaScript, directly out of the box. Furthermore, we have also implemented features for partially encrypted endpoints, URL sharing and session resumption. These functionalities utilize authenticated encryption, specifically AESSIV, to maintain security and integrity.

For transparency and accessibility, we will release our implementation open-source on GitHu[6]

5.1 RQ1: Security Discussion

In this section, we delve into the security ramifications of our proposed methodology, specifically crafted to curtail the scalability of bot operations. Unfortunately, directly quantifying the effectiveness

[6]https://github.com/8mas/encrypted-endpoints

of our strategy in thwarting bot proliferation presents a considerable challenge. This difficulty primarily stems from the inherent complexity in measuring the success rates of bots, as accurately distinguishing between legitimate and automated requests would essentially address the issue at hand. Moreover, we do not have direct access to popular web services that are frequently targeted by bot activities. Therefore, we opt for an analytical comparison, evaluating how our approach impedes the deployment of common bot creation techniques and attacker strategies, as outlined in our threat model detailed in Section 2.

Additionally, we conduct a comparative analysis between our approach and alternative bot mitigation strategies. In particular, we focus on code obfuscation due to its closely related nature.

5.1.1 Protection Against Scaling of Bots. A bot that scales is usable for other clients and accounts without being adapted to them beforehand. Exceptions are credentials or IDs, such as a username, password, or session token, which are needed to use the service.

As for the defense, we consider the following obfuscation approaches:

Code obfuscation: The client code (e.g., HTML, Binary, Android App) is obfuscated using, for example, techniques from related work [12, 41]. WWe describe the requirements for a suitable code obfuscation to be used in conjunction with our approach in Section 6.

Encrypted endpoints: The obfuscation approach described in this paper. Each request is on an encrypted URL.

Attacker: Endpoints Only. Code obfuscation cannot stop this attacker because the code is completely ignored. This attacker examines in which order to which endpoint what was sent and repeats the session. There are simple-to-use, automated tools that facilitate this [24, 34].

Encrypted endpoints stop this attacker completely. While the attacker can build a bot that works for their client, i.e., with their client identifier, the bot does not function for a different client. Furthermore, since this attacker cannot extract endpoints from the client-side application, the bot cannot be scaled. The attacker can also not predict or generate new valid URLs if a secure authenticated encryption method is used and correctly implemented.

Attacker: Endpoints and Data Parsing. Code obfuscation alone fails to offer any significant protection, as the endpoint-only attacker represents a specific subset of adversaries who can employ the same techniques for bypassing such measures. Similarly, encrypted endpoints, in isolation, do not guarantee resilience. Attackers might extract endpoints from client-side source code, such as Java classes in Android applications, by identifying the <a> tag's ID or XPath in HTML documents or using disassemblers for binary programs.

However, the combined application of code obfuscation and encrypted endpoints introduces a robust layer of protection. Encrypted endpoints specifically counteract the weak-points inherent in code obfuscation, by preventing straightforward access to the URLs by recording the traffic. Concurrently, code obfuscation complicates the attacker's ability to pinpoint and exploit individual URLs, thereby safeguarding against direct attacks on encrypted endpoints. It is crucial to ensure that endpoints do not inadvertently disclose information that could undermine this defense, for example, by adopting consistent encryption practices across URLs. Obfuscation becomes ineffective if URLs can be distinguished through their paths.

To augment this security strategy, we advocate for the use of advanced obfuscation techniques, akin to those described in [11], which can thwart brute-force attacks. Such attacks involve systematically probing all extracted URLs to identify the correct one through trial and error.

Direct attacks on encrypted endpoints, without compromising cryptographically secure methods, are implausible. Thus, attackers are compelled to extract URLs, with the complexity of this process being contingent upon the sophistication of the employed HTML obfuscation technique. Regularly updating the endpoints for each client can significantly escalate the effort and resources required for an attacker to maintain their offensive capabilities.

5.1.2 Encrypted Endpoints Compared to Other Approaches. Our approach does not claim to replace other approaches. On the contrary, it can and should be applied in addition. All following approaches are considered in the context of web bots and their scaling.

Code Obfuscation. The limitations of code obfuscation strategies have been discussed previously, both in the preceding section and in Section 3.2. These techniques enhance the difficulty of extracting specific data from client-side sources, such as from Java Classes or HTML, for instance, by employing XPath or CSS selectors (refer to Section 3.2). This increased complexity presents substantial obstacles for the development of bots that depend on information derived from the web, such as detecting product availability.

As described in Section 5.1.1, obfuscation methods fall short in mitigating the threats posed by API bots or tactics that involve replaying previously recorded sessions [24, 34]. Likewise, similar to our proposed model, they offer no defense against UI-based bots. However, encrypted endpoints effectively counter session replay attacks and escalate the difficulty for API bot operations. Although our methodology may be bypassed by parsing request responses to acquire currently valid endpoints, the integration of code obfuscation techniques significantly impedes the extraction of such information, including endpoints. This synergy underscores the advantage of combining our approach with code obfuscation methods.

User-Specific Client-Side API Keys. As discussed in Section 3.2, employing user-specific client-side API keys and CSRF tokens for user authentication and request validation marks preliminary measures against bot proliferation. Our strategy enhances these mechanisms, extending their application to comprehensively obstruct bot scalability through the mandatory retrieval and utilization of valid tokens for server requests.

CAPTCHAs. CAPTCHAs are another kind of defense. They are designed to confirm that a client is a human by their knowledge or ability to perform tasks. This means that they are an active challenge, which is counterproductive to gaining services [18], and wastes time [1]. Our approach does not hinder regular users, but it presents difficulties for those attempting to create or scale multiple bots. The difficulties arise that session replaying tools are rendered useless and force the extraction of client/account-specific encrypted endpoints. The use of code obfuscation techniques further complicates this extraction process.

Bot Detection and Risk Assesment. These approaches detect bots or gauge a client's likelihood of being a bot, leading to allowance, blocking, or CAPTCHA challenges. They aim to block bots without overusing CAPTCHAs, avoiding user frustration and revenue loss. Our approach does not differentiate bots. A possible combination with these approaches would be to provide only clients that are likely to be bots with encrypted endpoints and leave all others without.

Facebooks Encrypted URLs. As outlined in Section 3.2, Facebook's deployment of encrypted URLs primarily aims to prevent URL stripping, rather than deterring bot activities [3]. By encrypting and signing every URL parameter, the integrity of tracking parameters is preserved, thwarting attempts to remove them without compromising the link's validity. This method, while not designed for bot mitigation, shares similarities with our approach, prompting a detailed comparison in Table 2.

5.2 RQ2: Overhead

5.2.1 Ease of Integration. Integrating the proposed approach into a backend system is straightforward, facilitating the redirection of all requests through our middleware (cf. Listing 3). This middleware verifies the legitimacy of each request, offering configuration options such as custom identifier selection and support for partial URLs. For partially encrypted URLs, a custom validator must be written to check whether a partially encrypted URL is correct. Specifically, it must ensure that the dynamic, user-controlled part is valid so that the user cannot include more than necessary, such as extra parameters or paths. Load balancing and distributed deployment should function normally and seamlessly, as long as the client ID is deterministic, and the key derivation function (KDF) used to derive the client key is consistent, given that the implementation is stateless.

```
app.add_middleware(
middleware_class=EncryptedEndpointsMiddleware,
main_key=b"some_key")
```

Listing 3: Adding EncryptedEndpointsMiddleware to FastAPI

For the frontend, support is extended to Jinja2, enabling the invocation of an encryption function within our framework. This functionality is not confined to HTML but is also applicable to JavaScript tags and files (cf. Listing 4 and Listing 5).

```
TemplateResponse("start_page.html", {"request":
        request})
```

Listing 4: Rendering a Template Response in FastAPI

```
<!-- Encrypt URL to resources -->
<script src="{{encrypt_value('/templates/scripts.js',
        request)}}"></script>
<!-- Encrypt URLs -->
<i class="logout-icon"
        onclick="location.href='{{encrypt_value('/logout/',
        request)}}'"></i>
<!-- In JavaScript -->
<script>
fetch('{{encrypt_value("/posts/", request)}}')
</script>
```

Listing 5: Using Encrypted URLs in the Frontend

For binaries and applications, multiple integration methods exist. The simplest is to compile the binary with user-specific endpoints embedded. Alternatively, it is feasible to dynamically inject the endpoints into the application at startup. In this scenario, the application contacts a server, which then provides all necessary endpoints to the client. This necessitates that this information is obfuscated as well. For example, instead of transmitting the text form of endpoints, the endpoints could be embedded in a library, or the URLs could be transmitted encrypted and integrated into the main package within a secure enclave, such as Intel SGX.

5.2.2 URL Stretch. It is crucial to be aware of the limitations that various platforms impose on the length of URLs. Although the HTTP/1.1 specification RFC 2616 [15] does not define a maximum URL length, practical constraints are enforced by web browsers, for example, a limit of 2MB in Chromium[7]. Our experiments indicate that, as of the latest version in April 2024, all major desktop browsers (Firefox, Chrome, Edge, Safari), as well as mobile browsers (Chrome, Firefox, Safari), support URLs exceeding 20,000 characters.

Several factors influence the final size (L_{final}) of encrypted URLs, depending on the implementation specifics. The following holds for our implementation. These factors include padding ($O_{padding}$) to reach the block size, which might require an additional 128 bits, and the addition of a MAC (O_{mac}), contributing another 128 bits. Our implementation incorporates separators to support partially encrypted URLs due to their dynamic nature, adding an overhead of 2 bytes ($O_{separators}$). The overhead per encrypted URL segment, denoted by n_{blocks}, typically comprises only one segment, meaning that the entire URL is encrypted. Moreover, the transformation of encrypted data using base64 encoding typically increases the size by approximately 33% as it encodes every 3 bytes into 4 bytes (O_{base64}).

Given these considerations, the formula for calculating the length in bytes of data post-encryption can be expressed as follows:

$$L_{\text{final}} = L_{\text{data}} \cdot O_{\text{base64}} + n_{\text{blocks}} \cdot (O_{\text{padding}} + O_{\text{mac}} + O_{\text{separators}})$$

$$L_{\text{final}} = L_{\text{data}} \cdot 1.33 + n_{\text{blocks}} \cdot (16 + 16 + 2)$$

Given the length expansion, current browser limits and considering typical URL lengths, the length expansion from encryption does not appear to be problematic. Should one approach these limits, exploring text compression methods could offer a viable solution.

5.2.3 Runtime Performance. Table 5presents the latency results obtained within a local network environment using a test application we developed. The application returns a simple HTML page with a configurable number of embedded links. The baseline is the web application without our approach of encrypted endpoints, utilizing AES-GCM for authenticated encryption. It is worth noting that the processor in use is equipped with AES-NI [4]. The terminologies used in the table are defined as follows. *ENC* indicates a scenario where encrypted URLs are utilized, with the client key being dynamically derived for each request, thus eliminating the requirement

[7]https://chromium.googlesource.com/chromium/src/+/HEAD/docs/security/url_display_guidelines/url_display_guidelines.md

for the server to maintain state. *S-Key* denotes a method similar to ENC, wherein the client key is cached on the server, adopting a stateful approach to enhance performance by balancing storage space against CPU time. *S-URLs* employs a strategy akin to the S-Key method, with the encrypted URLs being cached on the server to avoid the need for recalculating or re-encrypting identical URLs.

The results from the first table highlight that caching the client key or the accessed URLs on the server, thereby maintaining a stateful approach (*S-Key*), reduces latency, though the overall runtime overhead remains minimal across all methods tested. Ideally, caching client keys temporarily, along with frequently accessed (hot) URLs, should be implemented to minimize latency.

	ENC	S-Key	S-URLs
Latency Increase	+0.0311ms	+0.0199ms	+0.0016ms

Table 3: Impact on backend server response times due to the handling (decryption) of encrypted URLs, measured in milliseconds. Profiling was conducted internally, avoiding end-to-end measurements due to their millisecond-level variability, which could compromise accuracy. An average end-to-end request in a local network is estimated between 12ms and 14ms. The average latency increase was determined over 100000 requests.

Table 4 examines the performance overhead of rendering encrypted URLs using Jinja2, employing profiling to ensure accuracy over end-to-end timing due to the latter's variability at the millisecond level.

The table compares the mean communication latency for the *base* approach, which involves not encrypting URLs, against various methods employing encrypted URLs. For each request, an HTML document containing between 10 to 100 unique URLs was returned. This comparison illustrates that caching both the client key and the URLs significantly reduces latency, a strategy that should be particularly effective for URLs with high demand.

	Base	ENC	S-Key	S-URLs
10 URLs	0.033ms	+0.158ms	+0.076ms	+0.023ms
100 URLs	0.087ms	+1.470ms	+0.662ms	+0.128ms

Table 4: Performance overhead of encrypting and rendering unique URLs. Profiling was utilized instead of end-to-end timing to circumvent the latter's millisecond-level variability, which could lead to inaccuracies. The table presents the mean communication latency in milliseconds for direct and encrypted URLs. Base refers to the baseline, meaning our web application without encrypted endpoints. The average latency increase was observed over 100000 requests.

5.3 RQ3: Protection Against Attacks

The methodology outlined in this paper primarily aims to mitigate bot-related threats while also functioning as an automatic whitelist mechanism for URLs. We briefly discuss this feature, considering it as a subject for future exploration.

This approach ensures that only URLs explicitly authorized by the server are accepted, offering protection against certain types of attacks:

Directory Traversal This method prevents attackers from navigating outside the web root directory to access restricted files, thus safeguarding against unauthorized directory access attempts (e.g., *example.com/../../etc/passwd*).

Accidental Data Exposure It protects against the accidental exposure of sensitive files, such as configuration files (*example.com/.env*), by ensuring that only URLs deliberately exposed by the server are accessible.

Local File Inclusion (LFI) It obstructs attempts to include files from the server's filesystem in the output of a web application, a tactic often used to execute arbitrary code or access sensitive information.

Reflected XSS and SQL Injection Attacks It offers some protection against attacks that misuse URL parameters to inject malicious scripts (Reflected XSS) or manipulate database queries (SQL Injection), as exemplified by attempts like *example.com?item='– drop table*. This method, however, is only applicable to endpoints where the URL parameters are predetermined, such as a shopping page displaying related products or a social media/blog platform suggesting tags (e.g., *example.com?tag=outfit*). This protection mechanism is ineffective if fully user-controlled input is accepted, such as in search fields. Additionally, in cases where this method is effective, it may inadvertently restrict users who guess parameters. For instance, a user interested in exploring hardware instead of outfits cannot simply change the tag in the URL bar, even if the tag exists.

At the heart of this protective mechanism is the requirement for each URL to carry a valid MAC issued by the server. The use of a MAC that resists existential forgery under chosen-message attacks ensures the impossibility for attackers to forge valid URLs.

However, it is crucial to recognize the limitations of this security measure. For URL segments that are completely under user control, such as parameters in search fields, our approach falls short. This is because the URL cannot be entirely pre-constructed by the server; it can only be partially created, as the user's search parameter is unknown. Further, in scenarios where an attacker successfully persuades the server to serve a malicious URL, the URL would be considered valid. In the previous example with the tag system (*example.com?tag=outfit*), this prevents an attacker from merely attempting to insert malicious payloads directly into the parameter system. However, if the attacker can create a new tag containing the malicious payload, the attack would succeed. The practicality of such an exploit depends on the website's specific configurations and security measures, which may deter or entirely prevent the attacker's success.

5.4 Limitations and Privacy Implications

While encrypted endpoints limit the scalability of bots, it is hard to quantify how much harder they make it. This is mainly due to the quality of the used obfuscation. In an ideal scenario, a perfect obfuscation is used in conjunction with our approach, which would prevent bots' scaling. However, there is no such thing as a perfect obfuscation.

Services that use encrypted endpoints and require URL sharing between users (e.g., an online shop or social media) or need to be interoperable with third-party services (e.g., a payment provider or identity provider) must identify URLs or URL paths that need to be shareable. These URLs and corresponding actions must be communicated to the middleware implementing the encrypted endpoints. In large applications, this might increase the initial cost of using encrypted endpoints.

Further, despite the introduction of unique endpoints for each account, attackers with substantial resources, be it in terms of computational power or time, could potentially overcome this hurdle. By programming bots that simulate user interactions with the UI, attackers could sidestep the account-specific endpoint requirement. While such UI-based bot creation might prove resource-intensive, especially in the case of smartphone apps requiring the use of Virtual Machines (VMs), determined attackers may still find ways to automate these processes effectively, e.g., by starting bots sequentially to save resources. Moreover, the reliance on UI interactions constrains the capabilities of these bots to the scope of functionalities exposed through the UI. This limitation could hinder attackers seeking more nuanced and intricate actions that direct controlled requests could achieve. Therefore, while the „encrypted endpoints" approach offers strong protection against simple and some advanced bots, it might fall short against adversaries who are adept at crafting UI-based automation techniques.

Our approach can be used more as defending against bots alone. It also destroys the ability to read URLs and their parameters. In today's Internet, the URL path or its parameters often reveal information about a resource. When these URLs are encrypted, this is no longer the case. Authenticating URLs alone cannot remove tracking parameters without invalidating the URL. Facebook has already started doing this [3]. Unfortunately, we do not have any technical countermeasures against this approach. The main problem with this approach, which makes it worse than tracking cookies, is that all tracking information is stored in the URL itself and thus affects shared links.

6 COMBINING ENCRYPTED ENDPOINTS WITH CODE OBFUSCATION

Code obfuscation is orthogonal to our approach but necessary to protect against advanced and realistic EP attackers. We define several requirements that a code obfuscation technique should fulfill. The issue is that even if all endpoints are encrypted, the location within the application remains unchanged. Thus, a bot creator does not need to save the endpoint but rather the position where the URL is stored. This could be a file offset, an XPath or CSS Selector in a web context, or even a line number in JavaScript.

We identified several requirements for a code obfuscation approach. The most important is the *obfuscation of the URL location*, ensuring that the URL is not easily identifiable through its position or the relationship between objects. The URL may also be split and stored in multiple locations if the context permits, such as in programming languages like JavaScript, but not in languages like HTML or CSS. Another requirement is *non-deterministic results*, meaning the code obfuscation should ideally produce different versions of the application for different users. Deterministic obfuscation methods should be combined with non-deterministic techniques. Additionally, the obfuscation should have *low overhead in runtime and build*. The resources required to build the obfuscated application should be minimal, which could be addressed by prebuilding multiple versions of the application. Lastly, *full application obfuscation* is essential, meaning the entire application containing or potentially containing URLs should be obfuscated. This is particularly important for web applications, where it is insufficient to obfuscate only the JavaScript if it inserts URLs into the HTML.

While most applications have many options due to their ability to modify themselves at runtime, standalone HTML pages without JavaScript present a challenge. HTML cannot modify itself at runtime and must therefore include all URLs in clear text. Given that the obfuscation technique can change the location of URLs as specified in our requirements, an attacker could still brute force all possible URLs. To mitigate this, some form of rate limiting or trap URLs should be included. For instance, if a bot attempts to access a fake URL, it should be blocked from further interaction.

Using our approach, the URLs do not reveal any information about themselves as they are encrypted and can be padded to equal lengths. Suitable approaches are discussed in Section 3.2.

7 CONCLUSION

Our investigation reveals that while numerous strategies primarily aim at complicating the initial creation of bots, our proposed methodology focuses on impeding their applicability across different user accounts. This objective is accomplished by assigning unique, encrypted endpoints (URLs) to each client, thereby necessitating that attackers extract fresh endpoints for every bot directly from server responses. Implemented as a middleware layer within the backend architecture, our solution facilitates seamless integration with existing systems, ensuring minimal impact on performance. An empirical evaluation of our approach indicates a modest increase in latency overhead by approximately 0.03ms - 0.0016ms for incoming requests under the most realistic operational scenarios. Alone, our method proves effective in thwarting the efforts of simple bots and automated tools. However, its efficacy is markedly enhanced when combined with targeted code obfuscation techniques, significantly curtailing the scalability of sophisticated bot operations. Looking forward, we aim to conduct a comprehensive empirical analysis of our approach, focusing on identifying and integrating code obfuscation methods specifically designed to combat bot activities. This will thereby further reinforce the security of web services against automated threats.

REFERENCES

[1] 2021. Humanity Wastes about 500 Years per Day on CAPTCHAs. It's Time to End This Madness. http://blog.cloudflare.com/introducing-cryptographic-attestation-of-personhood/

[2] 2022. 2022 Bad Bot Report | Evasive Bots Drive Online Fraud | Imperva. https://www.imperva.com/resources/resource-library/reports/bad-bot-report/

[3] 2022. Facebook Gets Round Tracking Privacy Measure by Encrypting Links. https://www.malwarebytes.com/blog/news/2022/07/facebook-gets-round-tracking-privacy-measure-by-encrypting-links

[4] 2023. Intel® Advanced Encryption Standard Instructions (AES-NI). https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-

encryption-standard-instructions-aes-ni.html

[5] Alejandro Acien, Aythami Morales, John V Monaco, Ruben Vera-Rodriguez, and Julian Fierrez. 2021. TypeNet: Deep learning keystroke biometrics. *IEEE Transactions on Biometrics, Behavior, and Identity Science* 4, 1 (2021), 57–70.

[6] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. 2015. Man-At-The-End attacks: Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications* 48 (2015), 44–57.

[7] Fatmah H Alqahtani and Fawaz A Alsulaiman. 2020. Is image-based CAPTCHA secure against attacks based on machine learning? An experimental study. *Computers & Security* 88 (2020), 101635.

[8] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 189–200.

[9] Alessandro Bessi and Emilio Ferrara. 2016. Social bots distort the 2016 US Presidential election online discussion. *First monday* 21, 11-7 (2016).

[10] BinBashBanana. 2024. BinBashBanana/html-obfuscator. https://github.com/BinBashBanana/html-obfuscator original-date: 2020-04-22T00:23:10Z.

[11] Douglas Brewer, Kang Li, Laksmish Ramaswamy, and Calton Pu. 2010. A Link Obfuscation Service to Detect Webbots. In *2010 IEEE International Conference on Services Computing*. IEEE, Miami, FL, USA, 433–440. https://doi.org/10.1109/SCC.2010.89

[12] Christian Collberg. 2001. *the tigress c obfuscator*. Retrieved 2021-12-07 from https://tigress.wtf/about.html

[13] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. 2018. Privacy Pass: Bypassing Internet Challenges Anonymously. *Proc. Priv. Enhancing Technol.* 2018, 3 (2018), 164–180.

[14] Ahmed Diab and Tawfiq Barhoum. 2018. Prevent XPath and CSS Based Scrapers by Using Markup Randomizer. *Int. Arab. J. e Technol.* 5, 2 (2018), 78–87.

[15] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. 1999. *Hypertext transfer protocol–HTTP/1.1*. Technical Report.

[16] R Fielding, M Nottingham, and J Reschke. 2022. RFC 9110: HTTP Semantics.

[17] Roy Fielding and Julian Reschke. 2014. RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): semantics and content.

[18] Nick Heath. 2010. Expedia on how one extra data field can cost $12m. https://www.zdnet.com/article/expedia-on-how-one-extra-data-field-can-cost-12m/. Accessed: 2021-10-18.

[19] Md Imran Hossen, Yazhou Tu, Md Fazle Rabby, Md Nazmul Islam, Hui Cao, and Xiali Hei. 2020. An Object Detection based Solver for {Google's} Image {reCAPTCHA} v2. In *RAID 2020*. 269–284.

[20] Apple Inc. [n. d.]. Replace CAPTCHAs with Private Access Tokens - WWDC22 - Videos. https://developer.apple.com/videos/play/wwdc2022/10077/

[21] ProtWare Inc. [n. d.]. Encrypt HTML source, Javascript, ASP. Protect links & images. HTML encryption. https://www.protware.com/

[22] Jscrambler. [n. d.]. Webpage Integrity: Manage Third-party Risks. https://jscrambler.com/webpage-integrity

[23] Timofey Kachalov. [n. d.]. javascript-obfuscator/javascript-obfuscator: A powerful obfuscator for JavaScript and Node.js. https://github.com/javascript-obfuscator/javascript-obfuscator

[24] Albert Koczy. 2023. Mitmproxy2swagger. https://github.com/alufers/mitmproxy2swagger

[25] Mohinder Kumar, MK Jindal, and Munish Kumar. 2022. A systematic survey on CAPTCHA recognition: types, creation and breaking techniques. *Archives of Computational Methods in Engineering* 29, 2 (2022), 1107–1136.

[26] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2014. Reducing web test cases aging by means of robust XPath locators. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, 449–454.

[27] Wei Liu. 2018. Introducing reCAPTCHA v3: the new way to stop bots. https://developers.google.com/search/blog/2018/10/introducing-recaptcha-v3-new-way-to. Accessed: 2021-05-20.

[28] Intuition Machines. 2018. Stop more bots. Start protecting user privacy. https://www.hcaptcha.com/. Accessed: 2021-05-20.

[29] Genesis Mobile. [n. d.]. JavaScript Obfuscator - Protect your JavaScript Code. https://jasob.com/

[30] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. 2021. Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 569–580. https://doi.org/10.1109/DSN48987.2021.00065

[31] Keaton Mowery and Hovav Shacham. 2012. Pixel perfect: Fingerprinting canvas in HTML5. *Proceedings of W2SP* 2012 (2012).

[32] August See, Leon Fritz, and Mathias Fischer. 2022. Polymorphic Protocols at the Example of Mitigating Web Bots. In *European Symposium on Research in Computer Security*. Springer, 106–124.

[33] August See, Tatjana Wingarz, Matz Radloff, and Mathias Fischer. 2023. Detecting Web Bots via Mouse Dynamics and Communication Metadata. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 73–86.

[34] see-aestas. 2020. Charles-Extractor. https://github.com/see-aestas/Charles-Extractor

[35] Suphannee Sivakorn, Iasonas Polakis, and Angelos D Keromytis. 2016. I am robot:(deep) learning to break semantic image captchas. In *2016 IEEE EuroS&P*. IEEE, 388–403.

[36] VMProtect Software. 2021. *VMProtect Software Protection*. Retrieved 2021-12-07 from https://vmpsoft.com/

[37] Grażyna Suchacka, Alberto Cabri, Stefano Rovetta, and Francesco Masulli. 2021. Efficient on-the-fly Web bot detection. *Knowledge-Based Systems* 223 (2021), 107074.

[38] Mahin Talukder, Syed Islam, and Paolo Falcarin. 2019. Analysis of obfuscated code with program slicing. In *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*. IEEE, 1–7.

[39] Oreans Technologies. 2022. Oreans Technologies : Software Security Defined. https://www.oreans.com/Themida.php Accessed 2021-12-07.

[40] Shardul Vikram, Chao Yang, and Guofei Gu. 2013. Nomad: Towards non-intrusive moving-target defense against web bots. In *CNS*. IEEE, 55–63.

[41] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. 2016. Webranz: web page randomization for better advertisement delivery and web-bot prevention. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 205–216.

[42] Ang Wei, Yuxuan Zhao, and Zhongmin Cai. 2019. A deep learning approach to web bot detection using mouse behavioral biometrics. In *Biometric Recognition: 14th Chinese Conference, CCBR 2019, Zhuzhou, China, October 12–13, 2019, Proceedings 14*. Springer, 388–395.

[43] Tara Whalen, Thibault Meunier, Mrudula Kodali, Alex Davidson, Marwan Fayed, Armando Faz-Hernández, Watson Ladd, Deepak Maram, Nick Sullivan, Benedikt Christoph Wolters, et al. 2022. Let The Right One In: Attestation as a Usable {CAPTCHA} Alternative. In *Eighteenth Symposium on Usable Privacy and Security (SOUPS 2022)*. 599–612.

[44] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2012. The power of obfuscation techniques in malicious JavaScript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*. IEEE, 9–16.

[45] Jason Yung. 2024. json2d/obscure. https://github.com/json2d/obscure original-date: 2016-05-30T22:04:01Z.

	Direct	ENC	S-Key	S-URLs
Latency 10 URLs	13.9ms	+3.1%	+2.4%	+1%
Latency 100 URLs	14ms	+4.3%	+2.1%	+0.7%
Latency 1000 URLs	14.1ms	+5.7%	+2.1%	+0.8%

Table 5: Mean communication latency in milliseconds relative to direct server communication, expressed as a percentage increase (N=100000). Dynamic URL generation and 100 KB webpage padding were employed.

A ALTERNATIVE PROXY IMPLEMENTATION

We further explored implementing our approach through a transparent proxy rather than as middleware, eliminating the need for any modifications to the service's code. While this method offered ease of integration to any backend, it introduced significant performance drawbacks, primarily due to the additional resources required for parsing.

A.1 Performance Overhead

Our proxy-based implementation involved intercepting requests to dynamically identify URLs, generate keys, and encrypt URLs, contributing to increased latency due to the computational demands of these operations. To quantify the impact on latency, we conducted tests on three simulated websites containing 10, 100, and 1000 URLs, respectively. Each website was standardized to 100 KB in size, inclusive of URLs and randomly generated padding bytes, with 1000 URLs approximating 95 KB of the total content. This choice was informed by an analysis of 10,000 random websites from the Majestic Million list, revealing a median presence of 106 URLs per site. The latency measurements were performed on a Debian 11 system equipped with an Intel i5-8365U CPU, targeting a Gunicorn/Flask HTTP server. Latency was calculated from the initiation of the request to the full receipt of the response, without reusing TCP connections for subsequent requests.

Table 5 presents the latency findings within a local network context, employing AES-GCM for authenticated encryption, with the processor supporting AES-NI.

The results indicate that direct server communication (*Direct*) is the fastest, with latency increases for encrypted URLs (*ENC*), stored client keys (*S-Key*), and stored encrypted URLs (*S-URLs*) being attributable to the additional computational steps involved. Notably, the performance gain from storing client keys or URLs is modest, given the already low overhead from encryption.

Transitioning from a proxy to backend middleware implementation enhances efficiency, albeit at the cost of requiring code adjustments. The proxy model, despite its straightforward setup, incurs significant performance losses, especially with an increasing number of URLs, due largely to the time-intensive nature of HTML URL parsing.

A.2 Organisational Overhead

The complexity of directly implementing encrypted endpoints varies with the existing codebase. Here, the primary challenge lies in statically extracting URLs from backend responses, a process that may miss URLs dynamically generated via JavaScript.

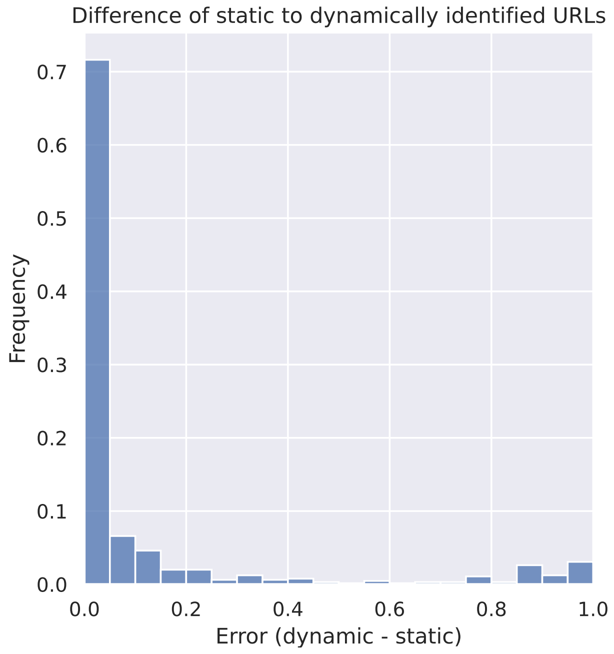


Figure 3: Overview of Possible Encrypted Endpoint Usage (N=10000)

Our analysis of 10,000 URLs from the Majestic Million list, comparing static and dynamic extraction methods, sought to quantify the potential discrepancy. While not exhaustive, this comparison sheds light on the limits of static extraction in capturing dynamically generated URLs, as illustrated in Figure 3.

This analysis reveals instances where dynamic extraction identifies URLs not found through static methods, with a mean error rate of 0.1188 and 78.22% of sites showing no discrepancy between the two. Further investigation into sites with large discrepancies highlighted asynchronous JavaScript processes as a primary factor, suggesting that encrypted endpoint deployment could benefit from mechanisms like mutation observers to capture and encrypt dynamically generated links.

Reflecting on the findings from Section 5.2, the deployment of our approach, particularly as a proxy, appears less burdensome than initially anticipated. The performance is deemed acceptable for non-time-critical applications, and the proxy model allows for experimentation without altering the existing system architecture.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009