Beyond Exploit Scanning: A Functional Change-Driven Approach to Remote Software Version Identification

Jinsong Chen[†*], Mengying Wu[†*], Geng Hong[†*], Baichao An[†], Mingxuan Liu[‡], Lei Zhang[†], Baojun Liu[§], Haixin Duan[§,¶] and Min Yang[†]
[†] *Fudan University, {jschen23, wumy21}@m.fudan.edu.cn, {ghong, bcan20, zxl, m_yang}@fudan.edu.cn*
[‡] *Zhongguancun Laboratory, liumx@mail.zgclab.edu.cn*
[§] *Tsinghua University, {lbj, duanhx}@tsinghua.edu.cn*
[¶] *Quancheng Laboratory*

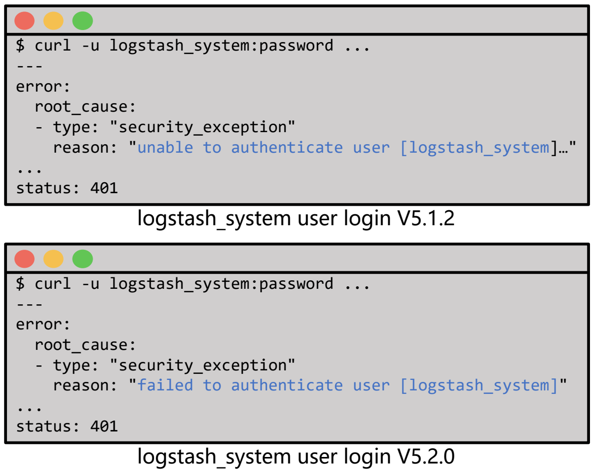
Abstract

Traditional attacks on remote software often fail to be armed with targeted software version information, leading to conspicuous brute-force attacks. Existing version identification tools, relying on predefined strings or patterns as fingerprints, can often not sketch software versions with defensive measures such as obfuscation or authentication.

This paper presents a covert and accurate version identification method based on noticeably different functional changes introduced by version updates. Our tool minimizes server noticeable probing behaviors by distilling domain knowledge from documents and change logs, and carefully designing dynamic probing sequences. We implemented and evaluated our prototype framework on Elasticsearch, Redis, Dubbo, Joomla, and phpMyAdmin, focusing on their versions from the past decade. Our tool achieved 2.8 times identification rates higher than previous works, with 65.37% fewer packages sent. Additionally, we conducted a large-scale scan of real-time data from Shodan and FOFA collected over two months, successfully identifying version information for 240,020 remote software instances, with 156,256 unrecognized by either platform. Our result reveals that over 72.25% users are still deploying versions released at least one year ago, facing significant vulnerability threats.

1 Introduction

Traditional attacks on remote software often rely on brute-force attempts, cycling through all available exploits to identify one that works [1]. However, this wastes significant resources due to the specificity of exploits to particular software versions [2] and risks alerting victims through repeated and conspicuous access attempts. Acquiring precise version information, therefore, becomes a critical multiplier for increasing the efficiency and stealth of such attacks. By narrowing the focus to vulnerabilities relevant to the target's software version, attackers can significantly enhance their chances of success



```
$ curl -u logstash_system:password ...
---
error:
  root_cause:
  - type: "security_exception"
    reason: "unable to authenticate user [logstash_system]…"
...
status: 401
```

logstash_system user login V5.1.2

```
$ curl -u logstash_system:password ...
---
error:
  root_cause:
  - type: "security_exception"
    reason: "failed to authenticate user [logstash_system]"
...
status: 401
```

logstash_system user login V5.2.0

Figure 1: A motivation example on Elasticsearch [3]. When logging in with the same username "logstash_system", versions v5.1.2 and v5.2.0 show different responses.

while reducing their likelihood of detection. Meanwhile, for security researchers, version information can also help them assess vulnerabilities and promptly alert developers.

Previous works have proposed several version identification methods based on predefined strings or patterns as fingerprints, such as keywords [4, 5], hash values of asset files [6, 7], and structures of HTML files [8]. However, Kondracki and Nikiforakis [9] found that they highly depend on the assumption that administrators do not modify or remove the content provided by "out-of-the-box" applications. Therefore, defensive measures, such as obfuscating version information in responses or enabling authentication, will significantly reduce their effectiveness.

To address the problem, we introduce a novel version identification method based on the noticeably different functional changes of softwares. Unlike previous modifiable features, *functional changes can hardly be modified, as they are the main causes that a software turns into a new version,* while

[*]These authors contributed equally to this work.

they will also lead to different responses than before. Figure 1 shows the response differences when logging into two Elasticsearch versions as the user "logstash_system". Starting from version 5.2.0, Elasticsearch introduced a new built-in user, "logstash_system". As a result, v5.1.2 treats the user as nonexistent, returning "unable to authenticate user {login_user}...", while v5.2.0 recognizes the user but fails to authenticate, returning "failed to authenticate user {login_user}". This case demonstrates that functional differences between versions can be leveraged to design probes that trigger distinguishable response variations.

There are two main challenges when developing a version identification tool based on software functional changes while minimizing server noticeable probing behaviors. First, effectively understanding and applying functional change requires specialized domain knowledge, which complicates the generation of probes to trigger these functionalities. Second, as users do not strictly use the "out-of-the-box" version, their responses may deviate from expectations. This poses a challenge in arranging backup probes that maintain accurate identification while minimizing their usage to remain covert.

For the first challenge, we supplemented release notes with contextual data from pull requests and user guides to acquire precise and adequate domain knowledge essential for effective probe design. For the second challenge, we proposed a probe planning solution using dynamic decision trees to address unexpected responses from customized content. Meanwhile, to resolve version conflicts caused by response spoofing, we use majority voting to determine the most likely version.

Laboratory experiments. We evaluated the method's effectiveness by implementing it on five widely used softwares with numerous vulnerabilities: Elasticsearch [3], Dubbo [10], Redis [11], Joomla [12], and phpMyAdmin [13], focusing on their versions from the past decade. The functional probes we generated can effectively distinguish versions where version strings are missing or security configurations are enabled, such as authentication and restricted external access. For example, among 191 Elasticsearch versions with default authentication enabled, 8 probes allow us to determine which of the 16 minor version ranges a server belongs to. Our tool significantly improves version identification accuracy across five software components compared to existing methods, with an improvement of up to 284%, while using 65.37% fewer probes. Moreover, only our tool can infer versions of Dubbo, as it does not expose any predefined string or pattern that reveals version information.

Real-world version scanning. We further conducted a large-scale measurement of five software systems in the real world. Specifically, using records from Shodan [14] and FOFA [15], we scanned 277,251 online servers for this software over two months. Our scanning tool identified version information for 240,020 servers, with 95.61% of the results achieving minor and patch-level identification. We obtained version information for 156,256 servers unrecognized by traditional device search engines. Our result reveals that over 72.25% users are still deploying versions released at least one year ago, and 24.98% Joomla users continue using decade-old versions. We also highlight the severe security risks posed by outdated versions and the absence of enabled security features.

Contributions. This paper makes the following contributions:

- We proposed a new version identification method based on noticeable functional changes introduced by version updates, which can minimize server noticeable probing behavior while guaranteeing accuracy and efficiency.
- We implemented our prototype framework, named *VersionSeek*, on Elasticsearch, Redis, Dubbo, Joomla, and phpMyAdmin. It achieved 2.8 times identification rates higher than previous works, with 65.37% fewer packages sent, especially when security features such as authorization are enabled.
- We conducted a large-scale, real-world scan on the five software and identified version information for 86.57% services, with 95.61% in minor and patch versions. Our result reveals that over 72.25% users are still deploying versions released at least one year ago, facing significant vulnerability threats.
- We open-sourced *VersionSeek* to help security researchers identify software versions and assess vulnerabilities.

2 Preliminary Study

2.1 Version Identification

Remote software version identification is typically performed through active scanning techniques by sending requests to targets and analyzing their responses. The two main techniques are banner grabbing via plain-text protocols, and multi-probe tools rely on multiple manually crafted requests. Banner grabbing connects to a service and retrieves version information from the welcome banner or specific handshake requests, such as "SSH-2.0-OpenSSH_7.6p1 Ubuntu-4ubuntu0.7." The multi-probe method manually collects basic interaction probes for various services to gather information, such as Nmap [16].

However, these methods highly rely on the assumption that administrators do not modify or remove the content provided by "out-of-the-box" applications [9]. They require the software to include at least one predefined string or pattern in its response that can be used to infer version information.

To understand whether current software provides such strings, we selected seven popular and representative open-source remote software systems, including a widely used content management system [17], web and terminal databases [3, 11], RPC frameworks [10] and web interface for SQL [13] with numerous vulnerabilities [18, 19, 20, 21]. Then we collect four software scanning tools: Nmap [16], Metasploit [22], BlindElephant [6] and WhatWeb [23]. By auditing their version identification code, we summarized the common types of predefined strings or patterns, as shown in Table 1.

Software	**Predefined String or Pattern**	**Feature Addition**	**Feature Adjustment**	**Feature Removal**		**Bug Fix**	**Optimization and Upgrade**		
		New Features	**Changes**	**Regressions**	**Deprecations**	**Bug Fixes**	**Performance**	**Dependency**	**Code**
Elasticsearch	V	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	-
Dubbo	-	$\surd$	-	-	$\surd$	$\surd$	$\times$	$\times$	$\times$
Redis	V	$\surd$	$\surd$	-	-	$\surd$	$\times$	-	-
Joomla	V,M,H,F	$\surd$	-	-	-	$\surd$	$\times$	-	-
Wordpress	V,M,H,F	$\surd$	$\surd$	-	$\surd$	$\surd$	$\times$	-	-
Drupal	M,H,F	$\surd$	-	-	-	$\surd$	$\times$	$\times$	-
phpMyAdmin	V,H,F	$\surd$	$\surd$	-	$\surd$	$\surd$	$\times$	$\times$	-

Table 1: Overview of a preliminary study on 7 open-source remote software about whether they support predefined string or pattern that reveals version information in the response and whether the response shows differently for each change type in release notes. In predefined strings or patterns, V represents version number, M represents magic pattern like URL recognition and HTML tags, H represents hash value, and F represents filenames. - means no such change found in release notes, $\surd$ means this change can result in different responses, and $\times$ means this change commonly does not result in different responses.

The result shows that not all software supports at least one type of predefined string that can be used to identify version information, such as Dubbo. Also, for web software, there are some banner obfuscation tools, such as ServerMask [24], which can be used to modify response headers or default banners to hide version information. Both situations can cause existing version identification tools to degrade in performance or fail to recognize the versions.

2.2 Functional Changes

Unlike previous modifiable features, *functional changes can hardly be modified, as they are the main causes that a software turns into a new version*, while they will also lead to different responses than before. As shown in Figure 1, when logging in with a specific username to two different versions of Elasticsearch, although both had authorization enabled and returned error messages, we could still distinguish between them based on their different responses. The reason for this difference lies in their implementation: version 5.2.1 introduced a new built-in user, "logstash_system".

Inspired by this example, we found that leveraging functional changes may solve the problems discussed in Section 2.1. Firstly, this method does not rely on predefined string or pattern in the response. Secondly, the differences caused by functional changes are not easily obscured or modified.

To examine functional changes across version updates and identify those causing response variations, we analyzed up to five versions per software, usually the first version of a new minor release. We reviewed release notes to categorize change types, selected two changes per type, and designed corresponding probes using domain knowledge from search engines and guidance documents. These probes were tested locally to verify if they triggered different responses.

Developers typically organize release notes according to the type of changes to make it easier for users to understand and apply. As shown in Table 1, following their conventions, we modeled the changes into the five categories:

- **Feature Addition**, refers to new functions or usages.
- **Feature Adjustment**, refers to adjusting existing functionalities in terms of support or usage.
- **Feature Removal**, refers to removing or deprecating outdated or unsafe features and feature rollbacks due to upgrades, such as deprecations and regressions.
- **Bug Fix**, refers to patching disclosed vulnerabilities or known issues.
- **Optimization and Upgrade**, includes code structure optimization, software dependency upgrades, and performance improvements.

The first three types of changes are mostly related to the core functions of the software. By adding, adjusting, or removing features, these changes better align with user needs, thus such modifications often result in observable differences in responses. Notably, although Bug Fix can also lead to response differences, using requests that trigger bugs violates ethical principles. Additionally, Optimization and Upgrade changes are primarily developer-oriented and rarely manifest in ways perceivable through user-accessible responses. Overall, we primarily select Feature Addition, Feature Adjustment, and Feature Removal as functional changes to construct probes.

2.3 Threat model

Our threat model assumes that an attacker can connect to and send probes to a host running known software. The host owner may take defensive measures such as obfuscating version, denying access to files containing leaked information, or using authorization or protection modes. The attacker aims to infer the software version running on the server.

Depending on the server's state, the attacker may infer the version at three levels: major, minor, and patch. Versions are typically represented in the format $x.y.z$, where x denotes the major version, which changes with incompatible API updates, y represents the minor version, which reflects backward-compatible feature additions, and z indicates the patch version, which accounts for backward-compatible bug fixes [25].

In closed-source and proprietary software, version updates are also accompanied by changes in functionality. The resulting variations in responses can thus be used to distinguish between different versions. However, the development information provided by such software is often limited or unavailable, making it difficult to automatically analyze their programs and extract functional changes. In contrast, given the public availability of version update information for open-source software and the accessibility of their source code, this paper uses open-source software as case examples for study.

3 Methodology

Based on the preliminary study, we design a novel version identification framework named *VersionSeek*, which consists of three modules: a functional probe generation module that extracts functional features from release notes and automatically generates probes that trigger these features, a response processing module that standardizes responses from different versions and classifies responses, and a version identification module that optimizes probe usage to maximize identification accuracy with minimal probes. Figure 2 illustrates the *VersionSeek* framework.

3.1 Functional Probe Generation

The functional changes introduced during the software update process are diverse, requiring specialized domain knowledge to fully understand and effectively apply. To generate probes to trigger these functional features accurately, we found that open-source software development often reflects community update needs, as reflected in pull requests (PRs). Also, user guides offer insights into how features are intended to be used. Combining this information, we can accurately locate the trigger probe of updated functions, as shown in Figure 3.

Functional Features and Context Collection. Each new version of a software typically includes Release Notes designed to assist developers in quickly understanding the features and functionalities of the update. These notes briefly outline the changes in the new version, typically presented as web pages and organized by the change types. As studied in Section 2.2, we leverage Feature Addition, Feature Adjustment, and Feature Removal as functional changes to construct probes. We used web crawlers to retrieve the original pages and parsed the HTML documents to extract the functional features.

While we can collect the target changes from release notes, feature descriptions in these notes are typically vague and lack sufficient context, making it difficult to construct specific probes. To address this, we collected additional contextual information from pull requests and the user's guide.

Pull requests (PRs) enable collaborative reviews of proposed changes, facilitating issue identification and improvement suggestions. Feature changes in open-source softwares are often documented in PR submissions, with descriptions linking to corresponding PR or issue numbers. By cross-referencing links or IDs in feature descriptions with records on the hosting platform and utilizing platform APIs, such as the GitHub API [26], we can obtain more detailed information about each feature, including functionality descriptions, code changes, and related discussions.

The user's guide typically provides detailed instructions and illustrative examples for each functionality. However, given the wide range of functionalities covered in the User's Guide, we need to locate the contextual information for each functional feature accurately. Here, by designing well-structured prompts, we guide the LLM in utilizing chain-of-thought(CoT) reasoning to retrieve the relevant documentation, as illustrated in Figure 3a. Specifically, by evaluating the similarity between document titles and functional features and using keyword frequencies as a double-check criterion, the most relevant documents from User's Guide will be retrieved.

Probe Generation. With the context of features, we can then manually construct specific functional probes. To automate this process, we adapted the Retrieval-Augmented Generation (RAG) method. As shown in Figure 3b, for each functional feature in the release note, we use the description in PR and the related user guide as additional information. At the same time, interaction examples are provided to standardize the format of the output. Finally, using the above contents as augmented information, LLM can automatically generate probes that trigger functional features.

Ethical Consideration. Considering that some probes may have side effects on target servers (e.g., configuration changes or brute-force attempts), we adopted measures such as Minimizing System Impact and Avoiding Brute-force-like Behavior to adhere to ethical guidelines. Details are provided in the Ethics Considerations section.

3.2 Response Processing

Our goal is efficient and covert version identification by selecting probes with optimal classification effectiveness, for which we designed a response processing module. By collecting responses from probes triggered across different versions and standardizing and classifying these responses, we can determine the version classifications for each functional probe.

Response Collection. To obtain response variations, we propose an automated deployment solution to streamline the deployment of diverse versions of software. Starting from the official source code, we observed minimal deployment differences between adjacent versions, enabling rapid deployment of multiple adjacent versions by modifying the version number. For distinct major versions (*e.g.,* Elasticsearch 6.x and 7.x), we selected representative samples from each major version to complete deployment and apply them to adjacent versions. Additionally, we adopted a containerized deployment to avoid port conflicts and dependency variations, packaging each software and its dependencies for cross-platform

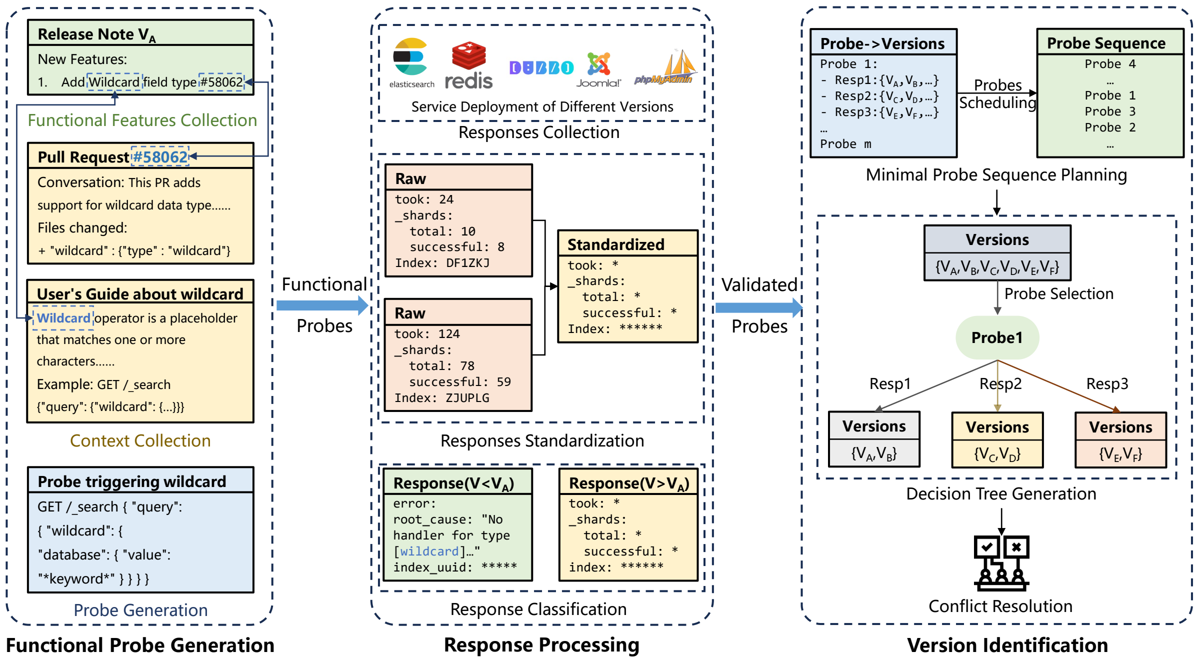
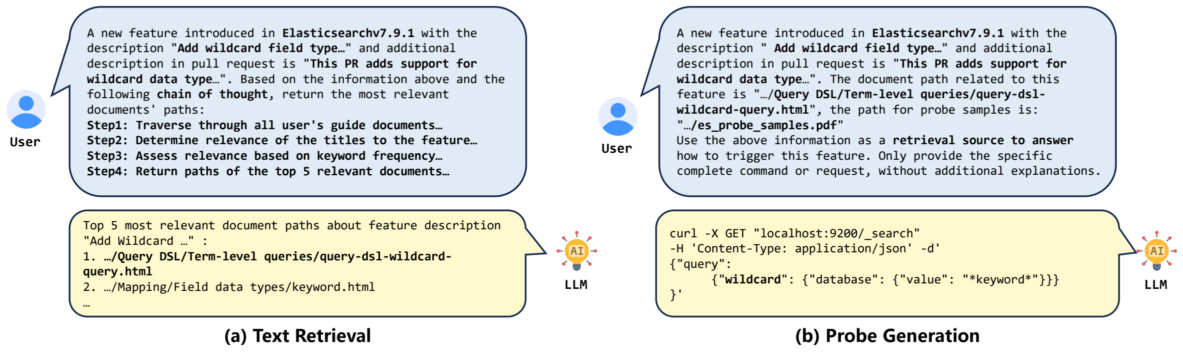


Figure 2: Overview of the *VersionSeek* framework.



```
A new feature introduced in Elasticsearchv7.9.1 with the
description "Add wildcard field type…" and additional
description in pull request is "This PR adds support for
wildcard data type…". Based on the information above and the
following chain of thought, return the most relevant
documents' paths:
Step1: Traverse through all user's guide documents…
Step2: Determine relevance of the titles to the feature…
Step3: Assess relevance based on keyword frequency…
Step4: Return paths of the top 5 relevant documents…
```

```
Top 5 most relevant document paths about feature description
"Add Wildcard …" :
1. …/Query DSL/Term-level queries/query-dsl-wildcard-
query.html
2. …/Mapping/Field data types/keyword.html
…
```

```
A new feature introduced in Elasticsearchv7.9.1 with the
description " Add wildcard field type…" and additional
description in pull request is "This PR adds support for
wildcard data type…". The document path related to this
feature is "…/Query DSL/Term-level queries/query-dsl-
wildcard-query.html", the path for probe samples is:
"…/es_probe_samples.pdf"
Use the above information as a retrieval source to answer
how to trigger this feature. Only provide the specific
complete command or request, without additional explanations.
```

```
curl -X GET "localhost:9200/_search"
-H 'Content-Type: application/json' -d'
{"query":
      {"wildcard": {"database": {"value": "*keyword*"}}}
}'
```

Figure 3: Overview of Functional Probe Generation.

flexibility. We then sent the saved probes to different versions of the software and recorded their responses.

Responses Standardization. Functional differences between versions result in distinct responses, however, various external noise factors can also lead to response variability even within the same version. For example, identical services operating in different network environments may produce different responses due to latency or bandwidth constraints. Furthermore, variations in operating systems or hardware configurations may further contribute to response discrepancies.

To reduce the influence of noise on response analysis, we focused on isolating information directly related to version-specific functionality. The response noise may vary across different types of software. To collect noise patterns for each software, we designed a differential testing method with three environmental factors: access times, varying data or content within the software, and test machines with different IP addresses. In each scenario, the same probe was applied to each software version five times, and responses were compared at the word level to identify discrepancies. This approach allowed us to derive noise-matching patterns specific to each software. We then matched the patterns and substituted them to standardize these noises within the responses.

Response Classification. For effective covert version identification, selecting probes with optimal classification performance is essential. To achieve this, we automatically group versions that produce identical responses, forming version classifications for each probe. This process provides valid probes and establishes a classification function to record the versions corresponding to different response types for each probe. The classification function can be formulated as $f(p_i, r_j) = \{v_a, v_b, v_c, ..\}$, given a valid probe p_i and its response r_j, it outputs a corresponding version set $\{v_a, v_b, v_c, ..\}$.

3.3 Version Identification

Since our goal is to identify the version of software, a single probe is insufficient for accurate identification. Therefore, we need to combine multiple probes to achieve maximum recognition effectiveness. To achieve covert and efficient version identification, we designed a dynamic decision tree algorithm to schedule the probes effectively. Figure 4 illustrates our dynamic decision tree method.

Problem Definition. We formulate the probe scheduling as the solution to the following problem:

- **Input:** A version set V to be distinguished, a set of available probes P, and the classification function $f(p_i, r_j)$.
- **Output:** A minimal probe sequence *sol* that maximally distinguishes V.

In the initial stage, V denotes the set of all versions of software, and P denotes all functional probes. However, in the real world, user-customized settings or content may cause some probes generated from local experiments to become ineffective, such as p_1 in Figure 4, requiring a re-scheduling based on the current state. Specifically, we record the remaining versions to be distinguished V_r and the unused probes P_r, re-plan a feasible solution for distinguishing V_r (Algorithm 2 in the appendix) and construct a identification decision tree (Algorithm 3 in the appendix) to guide the probe scheduling. As illustrated in Algorithm 1, this process is repeated until V_r cannot be further distinguished, which can be formulated as:

$$\forall p_i \in P_{\text{unused}},\ R_i \text{ is the response set of } p_i, \text{ and } \nexists r_j \in R_i$$

$$\text{such that } V_{ij} = f(p_i, r_j), \text{where } V_{ij} \neq \emptyset,\ V_{ij} \neq V_r, \text{ and } V_{ij} \subseteq V_r.$$

Minimal Probe Sequence Planning. To minimize the number of probes used while maximizing version identification, we employ a recursive reduction approach to find a locally optimal solution efficiently.

Firstly, we determine the minimal probe sequence required to uniquely identify each version. For a specific version v_a, we select the probe that can distinguish it from as many other versions as possible. This process is repeated until no further distinction is made. The selected probes are then stored in order, forming the minimal probe sequence corresponding to v_a. By performing the above operation for each version in V, we obtain a minimal probe sequence specific to each version, such as $v_a : [p_1, p_2, \ldots], v_b : [p_1, p_2, \ldots]$.

Secondly, we obtain an initial solution after merging the probe sequences for all versions. However, this solution may contain redundant probes, meaning that combining remaining probes can achieve the same distinction between versions. To address this, we recursively check for redundant probes and remove them until the removal of any single probe results in the inability to distinguish the version set maximally. The final probe sequence then becomes the optimal solution. Algorithm 2 in the appendix demonstrates the complete process.

Algorithm 1: Probe Scheduling

```
Data: Version set $V$, available probes $P$, host $H$
Result: Version identification result $V_r$
Function VersionIdentification($V, P, H$):
    $V_r \leftarrow V$;
    $P_r \leftarrow P$;
    while $V_r$ can be further distinguished using $P_r$ do
        $sol \leftarrow$ Minimal sequence from $P_r$ to distinguish $V_r$ (Algorithm 2 in the appendix);
        $tree \leftarrow$ Decision tree for $sol$ (Algorithm 3 in the appendix);
        foreach $p_{cur} \in tree$ do
            $P_r$.remove($p_{cur}$);
            $r_{cur} \leftarrow$ response from H using $p_{cur}$;
            $V_{cur} \leftarrow f(p_{cur}, r_{cur})$;
            if $V_{cur} = \emptyset$ then
                break;
            end
            $V_r \leftarrow V_{cur} \cap V_r$;
        end
    end
    return $V_r$;
```

Decision Tree Generation. Based on the minimal probe sequence, we constructed a decision tree, which stores the relationship between probe responses and versions, to guide the probe scheduling, as shown in Algorithm 3 in the appendix.

Initially, the root node is the set of versions V to be distinguished. The first probe in the minimal probe sequence, p_i, is then selected to create child nodes. Each branch represents a response type r_j, and the remaining versions, $V_{ij} = f(p_i, r_j) \cap V$, are treated as corresponding child nodes.

For each new child node V_{ij}, if no new probe can distinguish V_{ij}, it is designated as a leaf node. Otherwise, a new probe in the minimal probe sequence is selected, ensuring that this probe has not been used in the path from the root to the current node and that it can further distinguish V_{ij}. The same process for p_i is recursively applied to create new nodes, continuing until all branches terminate at leaf nodes, indicating that the remaining versions cannot be further distinguished.

Conflict Resolution. User-customized modifications can introduce version conflicts. For example, enabling a feature disabled by default in a newer version might cause the server to behave like an older version. To improve version identification accuracy, we apply majority voting algorithm, based on the observation that while user modifications might alter some features, most will still reflect the current version's characteristics. Figure 5 shows the conflict resolution.

When a version conflict arises, we backtrack the decision

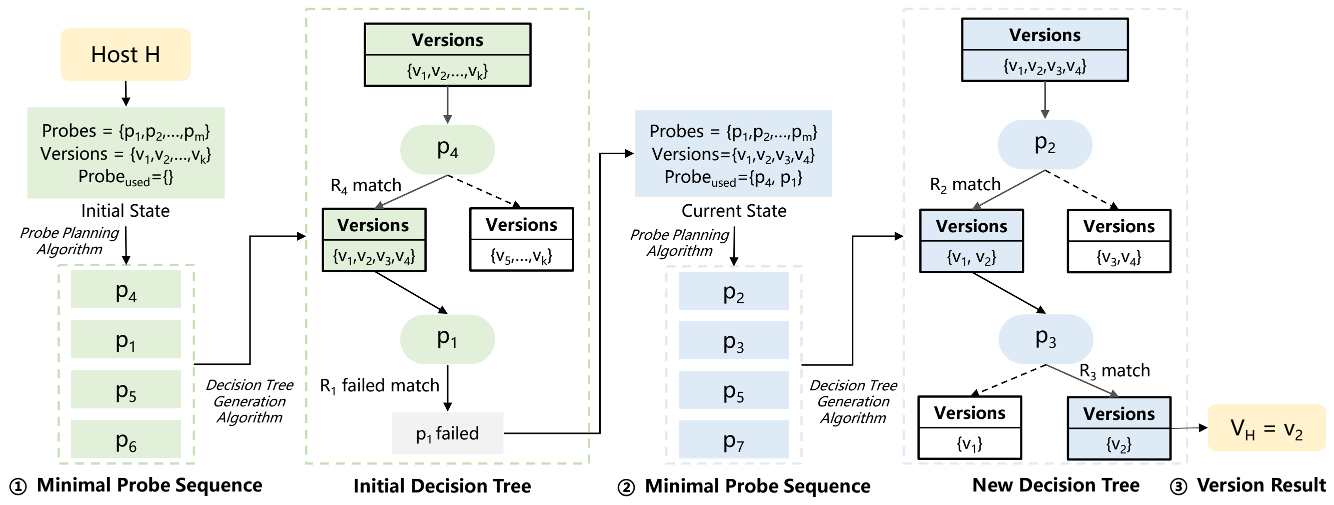


Figure 4: Overview of dynamic decision tree method for version identification. Algorithm 2 in the appendix focuses on minimal probe sequence planning, while Algorithm 3 in the appendix handles decision tree generation.

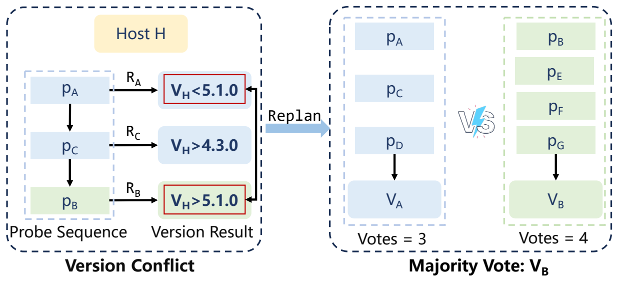


Figure 5: Overview of Conflict Resolution. Within the same probe sequence, p_A and p_B produced two conflicting version identification results. By selecting results supported by the majority of probes, we obtained high-confidence version identification results.

tree, compare the current probe's responses with others to identify inconsistencies, and record the relationship.

We use each conflicting probe as a root node to generate new decision trees, ensuring no conflicting probes are in the same tree. These trees are then sequentially used to identify the server until the probe limit is reached or no further distinctions are possible.

To balance overhead and accuracy, conflict resolution is limited to generating at most three new decision trees per scan. The probe usage limit for new trees is dynamically adjusted, and their cumulative usage does exceed that of the original decision tree, ensuring overall probe usage stays within acceptable limits. Subsequently, we compare the results of the new trees pairwise to identify non-empty intersections and merge them if applicable. For example, if p_B and p_C conflict with p_A but do not conflict with each other and have overlapping versions, we take the intersection as their result. Finally, we compare the path lengths of merged results and select the version with the longest path, as most probes agree with it.

4 Evaluation

To demonstrate the effectiveness and generality of our approach, five different types of software were selected, in which existing identification methods can be easily evaded, from the preliminary study: Elasticsearch [3], Redis [11], Dubbo [10], Joomla [12], and phpMyAdmin [13]. We implemented our prototype framework, named *VersionSeek*, based on their release versions over the past decade (2014-2024). Detailed implementation can be found in Appendix B. To evaluate the effectiveness of *VersionSeek*, we conducted the evaluation experiment driven by the following four research questions.

- **RQ1: (RAG Ablation Study)** What is the contribution of contextual information in the probe generation module to the experimental results?
- **RQ2: (Probe Effectiveness)** How effective are the probes generated by *VersionSeek*?
- **RQ3: (Scanning Efficiency)** What is the probe planning algorithm's contribution to *VersionSeek* 's runtime efficiency?
- **RQ4: (Version Identification Performance)** How effective is *VersionSeek* in identifying software versions compared to state-of-the-art tools?
- **RQ5: (Robustness)** How robust is *VersionSeek*'s when administrators deliberately conceal version information through techniques such as obfuscation or hiding?

4.1 Dataset

For RQ1, we manually analyzed a small-scale dataset derived from Elasticsearch. Specifically, we selected the initial and

Software	Version	# of Release-Notes	# of Pull Requests	# of Documents Retrieved	# of Probes Generated	# of Ineffective Probes	# of Valid Probes
ElasticSearch [3]	5.0.0-8.15.3	224	2,377	2,067	1,402	850	552
Redis [11]	3.2.0-7.4.1	107	671	438	341	207	134
Dubbo [10]	2.5.4-3.2.15	91	976	630	355	73	282
Joomla [12]	3.0.0-5.2.1	98	1,915	1,771	754	640	114
phpMyAdmin [13]	4.1.4-5.2.1	164	2,024	1,751	829	653	176
Total	-	684	7,963	6,657	3,681	2,423	1,258

Table 2: Summary of Document Collection and Probe Generation for four software.

final versions from each major version series (*e.g.,* 5.0.0 and 5.6.0 from the 5.x series) over the past decade. For each version, we chose up to 8 functional features. Using search engines and the Elasticsearch documentation, we manually constructed probes to trigger the corresponding functional feature changes. Finally, we obtained 30 functional features and their corresponding trigger probes. The dataset for RQ2 is derived from probes generated in our local experiments and the responses they triggered across different versions.

For RQ3 and RQ4, a dataset of many active servers annotated with known software type and version is required. To our knowledge, no available and reliable dataset currently exists*. We used Shodan to build a dataset annotated with software types, leveraging its continuous scanning of the internet, which provides real-time updates on the types and versions of services running on open ports. Specifically, we used Shodan's "product" filter to select records of specific software types and applied "before" and "after" filters to restrict the data collection to the most recent day. Then we used the Netcat [28] to scan open ports and filter out inaccessible or offline servers. Finally, we selected the latest records collected within a single day, and randomly chose 100 active servers of each software as the dataset.

The RQ5 dataset is derived from our locally deployed servers used to collect response variations, with two version-concealing methods applied, as detailed in Section 4.6.

4.2 RAG Ablation Study

Experimental Setup. To evaluate the impact of contextual information on probe generation, we guided the LLM to generate probes under two conditions: with context information (RAG) and without (Pure). To ensure a comprehensive assessment, we selected 4 popular LLMs based on the LLM Leaderboard [29]: GPT-3.5, Gemini1.5, Qwen2.5, and Llama3.2.

Study Results. Figure 6 shows the probe generation rate, *i.e.,* the number of correctly generated probes divided by the total number of functional features, under the two conditions. The improvement for all four LLMs with RAG, indicates that context and domain knowledge help LLMs understand the content of functional features and generate probes more accurately. Among them, GPT-3.5 showed the most significant improvement, raising its rate from 0.45 to 0.63. Qwen2.5 achieved the highest overall performance, reaching 0.67, and was selected for subsequent experiments.

*Although Joomla officially provides a showcase page [27], many servers were either offline or had migrated to non-Joomla services.

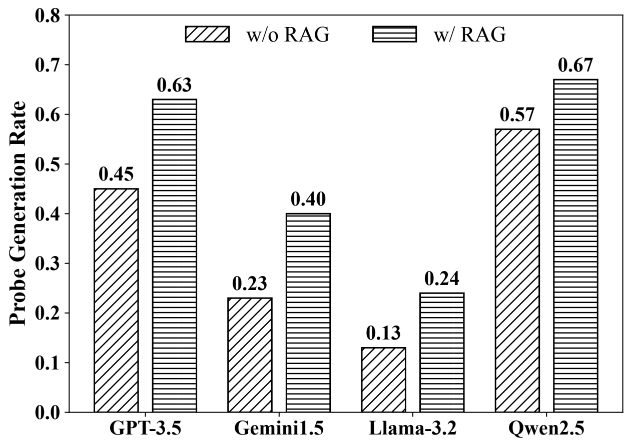


Figure 6: The probe generation rates of different LLMs under Pure and RAG conditions.

4.3 Probe Effectiveness

Table 2 shows the version we covered and the result of probes generated from these versions. We successfully retrieved the Pull Request discussions for 7,963 features. Based on the documents collected and CoT prompts, Qwen identified the relevant guidance for 6,657 functional features. By automatically matching standardized command templates, 3,681 probes were saved, all correctly formatted.

By automatically applying the ethical filtering rules outlined in Section 3.1 and testing the saved probes under various security configurations, we retained only those that produced distinguishing responses across different versions, as shown in Table 9 in the appendix.

Finally, we filtered out 2,423 ineffective probes that were either potentially harmful or exhibited poor classification performance, and obtained 552 valid probes for Elasticsearch, 134 for Redis, 282 for Dubbo, 114 for Joomla, and 176 for phpMyAdmin. The probe availability reached 34.17%.

Software	***VersionSeek***	**Nmap**		**Metasploit**		**Blindelephant**		**WhatWeb**	
		#	*Imp.*	#	*Imp.*	#	*Imp.*	#	*Imp.*
Elasticsearch	98	28	250.00%	64	53.13%	-	-	-	-
Redis	100	74	35.13%	75	33.33%	-	-	-	-
Dubbo	100	-	-	-	-	-	-	-	-
Joomla	98	-	-	81	20.99%	62	58.06%	66	48.48%
phpMyAdmin	96	-	-	-	-	62	54.83%	25	284.00%

Table 3: The version identification performance of *VersionSeek* and four tools across 100 servers for each of four services. *Imp.* indicates the improvement factor.

The number of response variants of a probe is related to its change frequency. We found that more than 88.01% probes can classify responses and versions into 2 to 5 categories. Additionally, some probes target frequently changing features, causing a higher number of categories. For example, the probe "GET /xpack/usage" in Elasticsearch, associated with the widely used xpack feature, changes frequently across minor versions, and can classify versions into 17 categories.

Even without authorization credentials, which typically result in error messages, the differences in error handling across versions still allow our probes to distinguish them. For example, the Elasticsearch probe "GET /_termvectors?fields=nested_field" returns an "Incorrect HTTP method" error in v6.8.0 - v6.8.23, while other versions report "missing authentication credentials." Additionally, due to the enhanced security measures introduced from version 8.0.0, their error message are more detailed compared to earlier versions, including v5.0-v5.6 and v7.1-v7.17.

4.4 Scanning Efficiency

We evaluated the runtime performance of *VersionSeek* during the local experiment, as shown in Table 7 in the appendix. We took 234 hours to generate all the probes, averaging one minute per functional feature using LLM. Although probe generation is time-consuming, it's a one-time process for each software. In actual software version identification, only probe scheduling and execution are repeatedly performed. Probe scheduling is typically completed within a few seconds, and the average execution time per probe is approximately 1.4 seconds.

Table 7 in the appendix presents the average scanning overhead of *VersionSeek* on real servers. When scanning all locally deployed versions of the services, we identified each version using very few probes. Specifically, *VersionSeek* required an average of 5.8 probes for Elasticsearch, 4.867 for Dubbo, 2.633 for Redis, 6.233 for Joomla, and 9.416 for phpMyAdmin. Compared to scanning a Joomla website with WhatWeb in aggressive mode, which typically requires sending 18 requests, we achieved 65.37% fewer packages sent. And we can complete the scanning in 7.4 to 14.611 seconds on average. This demonstrates the feasibility of conducting large-scale version identification measurements using *VersionSeek*.

4.5 Version Identification Performance

Baselines and Experimental Setup. We systematically collected software version identification tools by searching keywords such as "software/service version identify/infer/detect" and expanded our collection by exploring citation relationships. Ultimately, we collected eight tools: Wappalyzer [30], VersionInferrer [31], WhatWeb [23], BlindElephant [6], Nmap [16], Zgrab [32], Nmap Scripting Engine [33], and Metasploit [22].

The first four tools [6, 23, 30, 31] are designed for web applications, making them ineffective for identifying non-HTTP(S) services. However, since Wappalyzer transitioned from open-source to closed-source and VersionInferrer depends on Wappalyzer's files, we selected WhatWeb, BlindElephant, and Metasploit-Joomla for comparison. In contrast, widely used scanning tools like Nmap and Zgrab support multiple protocols and services. Additionally, Metasploit and the Nmap Scripting Engine (NSE) include numerous exploits for various software. We analyzed their source code and extracted only those components specifically designed for version identification. Code analysis shows that NSE, Metasploit, and Zgrab use similar methods to identify Elasticsearch, Redis, and Dubbo. We selected Metasploit as a representative for comparative analysis.

All comparison tools are configured for optimal performance. Specifically, BlindElephant is preset with the application software type, while both WhatWeb and BlindElephant operate in aggressive mode. Besides, both BlindElephant and WhatWeb were equipped with the latest 2024 fingerprint database, based on the source code provided in Kondracki and Nikiforakis [9].

To enable a detailed comparison between *VersionSeek* and these modern techniques, we implemented both approaches on phpMyAdmin, a representative target, as existing tools provide comparatively more comprehensive fingerprint databases for this software. Specifically, to construct a training dataset, we utilized 15 probes from BlindElephant, which are the default probes used for identifying the phpMyAdmin version, along with their corresponding responses collected from 164 locally deployed instances of different phpMyAdmin versions. Based on this dataset, we implemented two version identification tools: one leveraging a random forest algorithm to train a machine learning–based classifier, and the other employing black-box differential analysis to construct a decision tree for version identification.

Compared to existing tools. Table 3 presents the results of evaluating *VersionSeek* and four existing tools on the online servers described in Section 4.1. *VersionSeek* demonstrates significant performance improvements over all four tools, notably a 284.00% increase in accuracy for phpMyAdmin compared to WhatWeb. For Dubbo, existing tools detect only the service type via the "dubbo" string, while our tool successfully identifies their versions.

Nmap uses its regex-based nmap-service-probes [1] database for version detection, but performs poorly on Elasticsearch (28%) due to flexible JSON fields and dynamic values causing mismatches. Metasploit parses response content based on protocol specifications, achieving better performance on Elasticsearch (64%) than Nmap. When authentication is enabled, both Nmap and Metasploit fail due to a mismatch with the expected predefined string or pattern. In contrast, *VersionSeek* can identify versions even when authentication is enabled or access is restricted. For instance, we identified minor versions of 32 Elasticsearch servers and patch versions of 4 Elasticsearch servers, all with authentication enabled. We also identified minor versions of 5 Redis servers with authentication, as well as major versions of 16 Redis servers and minor versions of 4 Redis servers, all of which denied external access.

Unlike the previous two, the default handshaking response of Dubbo has no version numbers, and lacks commands like "INFO" to directly obtain version information. Therefore, neither Nmap nor Metasploit could identify any Dubbo server versions. Our tool, however, identifies versions by updated functional changes in commands such as “invoke” and “help”. On 100 Dubbo servers, *VersionSeek* identified 46 versions at the patch level, 49 at the minor level, and 5 at the major level.

Existing Joomla version identification tools, such as those using asset file hashes, are increasingly ineffective as administrators adopt countermeasures. For instance, 17 websites blocked access to version-revealing paths like “language/en-GB/en-GB.xml”, causing Metasploit-Joomla to fail in version identification. In contrast, *VersionSeek* leverages functional probes for version identification. For example, the “htaccess.txt” probe is a configuration file preconfigured by Joomla with security guidelines and environment checks, changes across versions due to updates in deployment and configuration. Finally, *VersionSeek* successfully identified minor versions of 8 websites and patch versions of 90 websites.

Similar to Joomla, existing phpMyAdmin version detection tools highly rely on predefined string patterns and hash-based matching. However, these rule sets are typically public, allowing administrators to block corresponding paths such as “/doc/html/setup.html” and “/ChangeLog”. Such measures were adopted by 18 websites, causing BlindElephant to fail. Differently, *VersionSeek* infers version by analyzing functional differences, such as variations in supported languages on default pages, which are typically accessible. As a result, *VersionSeek* identified major versions of 3, minor versions of 6, and patch versions of 87 phpMyAdmin servers.

Overall, excluding eight servers running versions not covered by *VersionSeek*, based on functional probes, *VersionSeek* has good performance on all five software. Even with security mechanisms enabled or when responses lack explicit version information, our tool identifies versions by analyzing functional differences, successfully recognizing 21 major, 107 minor, and 50 patch versions that might otherwise be overlooked by other tools.

Compared to modern methods. We evaluated the black-box differential and ML-based methods on 100 real-world phpMyAdmin servers, which identified 73 and 59 versions, respectively. *VersionSeek* achieved superior performance, with improvements of 31.50% and 62.71% over the two methods.

This performance gap can be attributed to both black-box differential and ML-based approaches using publicly available probes from BlindElephant, which may be blocked by administrators. We observed that 18 servers had implemented such defenses, rendering these methods ineffective.

In contrast, some functionality-based probes remain effective. For instance, essential pages like index.php, which are rarely blocked, expose version-specific differences (e.g., version 4.9.2 introduced Thai support, reflected by an additional language option on the interface), enabling *VersionSeek* to infer version. Besides, compared to *VersionSeek*, requiring 9.41 probes on average, the ML-based method incurs higher request overhead, as it always sends 15 probes to each target to construct fixed-size input vectors from the responses. Although the black-box differential method employs a decision-tree-based approach that reduces probe usage to some extent, its identification performance degrades when a probe fails, as it lacks dynamic decision tree generation, preventing the selection of an effective alternative probe. This limitation often arises in real-world settings where user-customized configurations may render certain probes ineffective.

4.6 Robustness of Version Identification

Methodology. We simulate adversarial scenarios by applying two version concealing techniques: obfuscation and hiding. Based on our preliminary study, except for Dubbo, all other studied software provide version information through specific paths or commands, as summarized in Table 4. In the obfuscated scenario, version strings in responses are replaced with placeholders, *i.e.,* x.x.x. In the hiding scenario, requests to

Software	Commands / Paths
ElasticSearch [3]	/
Redis [11]	INFO
Joomla [12]	/administrator/manifests/files/joomla.xml, /language/en-GB/en-GB.xml
phpMyAdmin [13]	/README, /ChangeLog

Table 4: Version Disclosure Commands or Paths.

version-leaking paths or commands are blocked entirely.

Specifically, by employing proxy-based request interception on our locally deployed servers, we constructed this adversarial dataset to evaluate the robustness of version identification, comparing *VersionSeek* against existing tools.

Result Analysis. The experimental results are shown in Table 8 in the appendix. Under both adversarial scenarios, *VersionSeek* achieves 100% version identification rate across four software by leveraging functional response differences rather than relying on predefined version patterns such as version strings. In the obfuscation setting, version-specific changes remain observable outside the version string. For instance, starting from Redis version 6.2.0, the INFO response includes a new field, "server_time_usec", which helps distinguish it from earlier versions. In the hiding setting, when probes fail due to altered responses, *VersionSeek* automatically generates and evaluates subsequent decision trees to continue inference. In contrast, tools such as Nmap and Metasploit, which rely heavily on explicit version strings, fail to identify Redis and Elasticsearch versions under both conditions.

Although BlindElephant demonstrates a certain level of robustness for Joomla and phpMyAdmin (29%, 100%), it lacks a dynamic scheduling strategy and consistently sends a fixed set of 15 probes to each server.

In summary, *VersionSeek* shows strong robustness in adversarial settings by identifying versions through functional differences rather than explicit version disclosures. Furthermore, its dynamic decision tree enables adaptive probe selection based on runtime responses, achieving accurate identification with fewer requests than traditional hash-based methods.

5 Real-World Measurement

Based on laboratory experiment results, we developed a version scanning tool to evaluate the distribution of vulnerable softwares in real-world scenarios. To adhere to ethical guidelines, we implemented several measures, such as minimizing system impact and avoiding brute-force–like behavior. Further details are provided in the Ethics Considerations section.

5.1 Dataset

Due to the time required for scanning all open devices on the internet and potential ethical concerns, we utilized Shodan and FOFA real-time updated scan results as the data source because they employ the most aggressive scanning strategy compared to other well-known device search engines [34].

We crawled real-time updated data from Shodan and FOFA over a two-month period, collecting records that include host and port information, response data, and identified service types. According to our analysis, among the five studied software systems, only a subset of Redis and Elasticsearch records contain explicit version information.

To mitigate potential bias from the presence or absence of version information in server records, we used platform-specific search syntax to construct targeted queries that obtained both versioned and non-versioned records for Redis and Elasticsearch, as well as records for the other three software. As shown in Table 5, we collected 60,315 distinct ElasticSearch servers, 178,477 Redis servers, 10,114 Dubbo servers, 150,776 Joomla servers, and 75,599 phpMyAdmin servers.

Software	Shodan		FOFA		Total
	w version	w/o version	w version	w/o version	
Elasticsearch	18,893	13,107	21,947	6,368	60,315
Redis	83,015	17,375	42,284	35,803	178,477
Dubbo	-	5,205	-	4,909	10,114
Joomla	-	106,607	-	44,169	150,776
phpMyAdmin	-	45,536	-	30,063	75,599
Total	101,908	187,830	64,231	121,312	475,281

Table 5: Data record across five software retrieved from Shodan and FOFA.

Since these records are time-sensitive, some servers were offline or had connection timeouts during our scans. As shown in Table 6, we successfully identified the version for 240,020 servers out of 277,251 active ones. Among these, 191,102 servers lacked any version information. For the remaining 86,149 servers with reported versions, our tool achieved an identification rate of 97.23%. Failures were mainly due to outdated versions not included in our database. Other contributing factors included service migration, request denials, and honeypots, which are further discussed in Section 5.3.

Most of our identification results are at the minor(27.33%) and patch levels (68.29%), and since software vulnerabilities typically occur within specific version ranges, our results are enough to match the most relevant vulnerability databases. Then we assessed vulnerability threats by matching the corresponding CVEs. Specifically, we used the CVE Details API [35], requesting and obtaining vulnerability information for each version, including CVE ID, description, severity, and other fields we use for further analysis.

5.2 Results

To illustrate the potential risks associated with overlooked version information, we mapped the identified version results from our tool to their corresponding CVEs. As our functional

Software	# of Server	Major	Minor	Patch
Elasticsearch	34,631	455	8,332	25,844
Redis	90,245	9,919	26,409	53,917
Dubbo	4,671	141	1,999	2,531
Joomla	70,215	12	24,827	45,376
phpMyAdmin	40,258	2	4,020	36,236
Total	240,020	10,529	65,587	163,904

Table 6: Real-world large-scale version identification results.

change-based version identification tool returns a list of versions, potentially spanning one or more minor version ranges, we use the union of CVEs associated with each version as identified potential vulnerabilities. Figure 7 shows the version distribution and the corresponding CVEs for five software.

Elasticsearch. We identified version information for 21,162 open and 13,469 authenticated Elasticsearch servers. Figure 7(a) reveals that the most prevalent server version range (26.84%) 7.14.0–7.17.12 (August 2021 – July 2023), is vulnerable to 10 CVEs, including 5 high-risk vulnerabilities (e.g., privilege escalation, denial of service). The most critical one is CVE-2023-31418 [36], which was patched in versions 7.17.13 and 8.9.0, potentially affecting 27,035 servers, of which 10,252 are authentication servers.

Compared to Shodan and FOFA, *VersionSeek* identified versions for 13,153 more servers, of which 12,950 require authorization. The versions of these authorized servers are mainly concentrated between 6.8.0 and 8.8.2, totaling 9,687 servers. A possible reason for this trend is that starting with version 6.8.0, Elasticsearch introduced free security features, such as role-based access control, to manage user access [37].

Redis. We identified 60,139 open, 19,265 requiring authentication, and 10,841 Redis servers that denied remote access. As shown in Figure 7(b), versions 7.4.0–7.4.1 (28.29%) are the most prevalent, released after July 2024, and only related to two CVEs. Besides, only 6.27% have enabled authorization among these servers, while the remaining 93.73% are at significant risk of security threats.

Compared to Shodan and FOFA, *VersionSeek* identified 16,370 more servers requiring authorization and 10,841 servers denying external access. Although deny-type servers return denied messages for any request, the responses vary across versions due to functional changes, as shown in Figure 8. Versions before 7.0 emphasize “bind address” while versions after 7.0 only emphasize “set password for default user”. Additionally, version 7.0 contains a typo ("setup a an"), which was fixed in version 7.2. These observations allow us to categorize the deny servers into three types. Among them, 7,460 (68.81%) are running versions earlier than 7.0, indicating some users with older versions recognized the insecurity of not setting a password and thus restricted external access.

Dubbo. We identified the Dubbo version for 4,671 Dubbo servers out of 4,711 online ones. Figure 7(c) shows that versions 2.5.5–2.5.10 (last updated 2018, 23.83%) and 2.6.8–2.6.12(last updated 2021, 21.17%) are most prevalent, associated with 9 and 8 CVEs, respectively. Eight of these CVEs are critical, mainly related to deserialization vulnerabilities that lead to remote code execution. Furthermore, we observed that 855 servers (18.30%) were running minor version 3.2, released in April 2023. The versions are associated with significantly fewer known CVEs, highlighting the importance of timely updates in mitigating vulnerabilities.

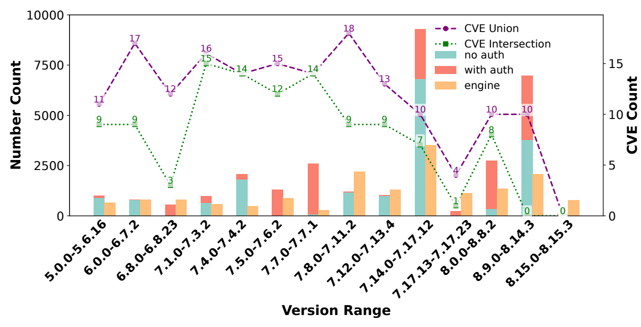
Joomla. We identified the Joomla version for 70,215 servers out of 77,272 online servers, while the remaining servers rejected connection requests. Figure 7(d) shows that the most popular minor version is 3.9, with 17,077(24.32%) servers. 62.28% of these servers are in the version range 3.9.2–3.9.10, last updated in July 2019, and potentially affected by at least 73 CVEs. Among these, the most severe are CVE-2019-19846 [38] and CVE-2022-23797 [39] (CVSS 9.8), which affect 39,148 (55.75%) and 47,865 (68.17%) servers, respectively, highlighting the significant vulnerability threats Joomla faces in real-world scenarios. Besides, 17,538 (24.98%) of deployed Joomla versions are lower than 3.3.0, last updated in April 2014, indicating that some users still persist in using outdated versions from a decade ago or earlier.

phpMyAdmin. We identified the phpMyAdmin version for 40,258 servers out of 59,073 online servers. Figure 7(e) shows that the most popular minor versions are 5.1.0–5.1.4 (last updated in May 2022, 44.31%) and 5.2.0–5.2.1 (last updated in Feb 2023, 35.47%), associated with 7 and 3 CVEs, respectively. The most severe among them is a CRITICAL SQL Injection vulnerability (CVSS 9.8), CVE-2020-22452, which affects versions 5.0.0–5.1.4 (48.40%). Earlier versions are affected by more vulnerabilities, but their adoption rate remains low, accounting for only 16.14% of deployments.

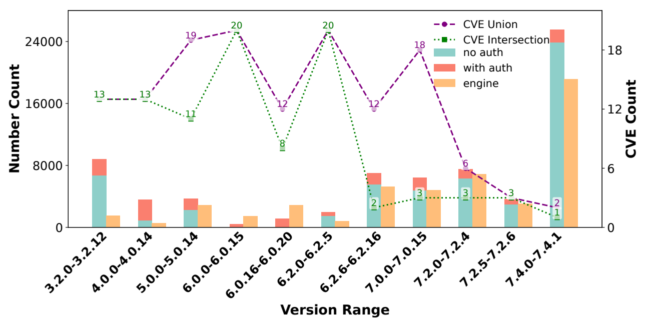
Overall, our measurement reveals that for each remote software, a significant portion (79.57%, 71.71%, 81.70%, 73.14%, 64.53%) of Elasticsearch, Redis, Dubbo, Joomla and phpMyAdmin users are still deploying versions released at least one year ago or earlier, accounting for 72.25% of all measured services. The most severe case is Joomla, where 24.98% of users continue using decade-old versions. This highlights that these alive remote services remain at significant risk of vulnerabilities. Besides, enabling the security measures provided by the software can significantly reduce potential threats and prevent some untrusted scanning or identification activities.

5.3 Case Study

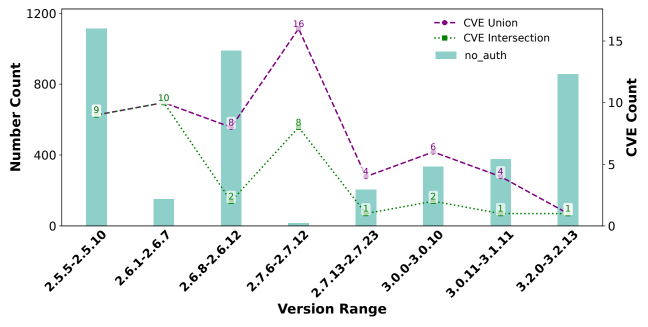
During large-scale scanning, we found some misreported results from Shodan, *i.e.,* honeypots, which are deliberately designed systems used by researchers to lure malicious attackers and analyze their techniques. We identified some servers suspected to be honeypots, which exhibited two key characteristics: incomplete function and unchanging responses.



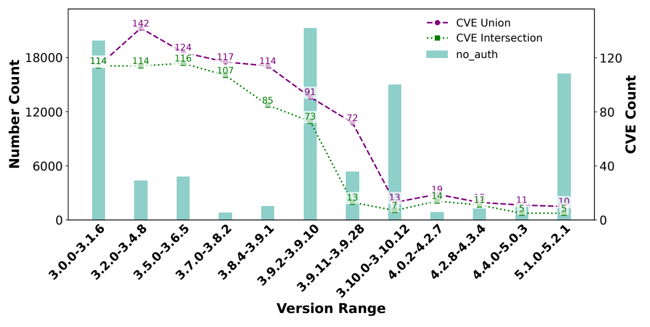
(a) Elasticsearch



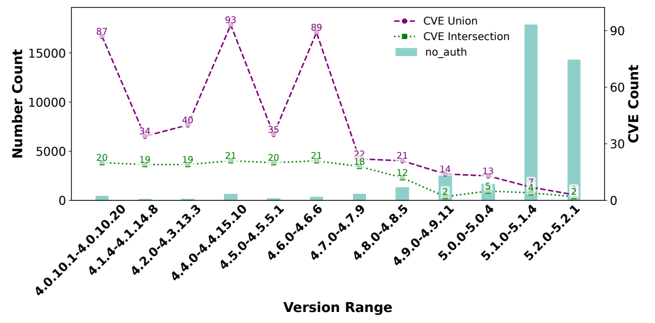
(b) Redis



(c) Dubbo



(d) Joomla



(e) phpMyAdmin

Figure 7: Distribution of version ranges and their corresponding number of CVEs for Elasticsearch servers (with and without authentication, and found by device search engines like Shodan), Redis servers (with and without authentication, and found by device search engines like Shodan), Dubbo, Joomla and phpMyAdmin servers. We merged our results at minor version level.

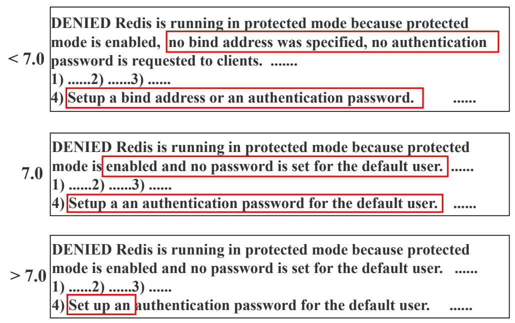
An example of incomplete function includes servers failing to correctly parse command arguments in Redis. When sending a request like "NOTCOMMAND ARGS", older Redis versions typically return an error message like "ERR unknown command NOTCOMMAND". Compared, newer versions append additional details, *e.g.,* "with args beginning with ARGS'. Some servers mimic the response of newer versions but fail to parse the arguments correctly. Their response is like "ERR unknown command NOTCOMMAND, with args beginning with:" without parsing "ARGS," revealing inconsistencies in their functionality. Also, we found some servers identified as Elasticsearch by Shodan consistently returned the default response irrespective of the request parameters or path, exhibiting unchanging responses.

These honeypot examples show that many version identification tools, which rely on fixed requests to obtain responses, can be easily deceived by fabricated simple responses. In contrast, *VersionSeek*, based on functional changes, achieves efficient and accurate version identification by dynamically scheduling the optimal sequence of probes.

6 Discussion

6.1 Security Implication

Our research proposes a novel approach to version identification, leveraging functional changes to distinguish between different versions based on subtle response variations that are



```
DENIED Redis is running in protected mode because protected
mode is enabled, no bind address was specified, no authentication
password is requested to clients.  .......
1) ......2) ......3) ......
4) Setup a bind address or an authentication password.        ......
```

```
DENIED Redis is running in protected mode because protected
mode is enabled and no password is set for the default user. ......
1) ......2) ......3) ......
4) Setup a an authentication password for the default user.  ......
```

```
DENIED Redis is running in protected mode because protected
mode is enabled and no password is set for the default user.   ......
1) ......2) ......3) ......
4) Set up an authentication password for the default user.    ......
```

Figure 8: Responses from different minor versions of the Redis server that deny external access.

less likely to be modified or detected. Our tool demonstrates better scalability, applicability, and robustness compared to existing version identification tools.

Our measurement reveals that many users still rely on outdated servers and fail to enable the security measures provided by the software, exposing them to vulnerability threats. Compared to the results of state-of-the-art device search engines, we identified and assessed a broader attack surface that was previously overlooked due to missing version information.

6.2 Limitation

Although our *VersionSeek* achieved the best remote version identification performance, we acknowledge that our approach has limitations. First, the probe generation of *VersionSeek* relies on the testing of locally deployed components. While applicable to all versions, source code and dependencies were hard to collect for many older versions, thus, we focused on the past decade. Missing deployment environment of some versions within this timeframe also results in imprecise version identification for them. However, the expected low prevalence of services consistently running these missing versions minimizes their impact on the overall results.

Secondly, *VersionSeek*'s effectiveness is reduced against authenticated components, as many probes require authentication. However, *VersionSeek* can still infer approximate version ranges even in these cases based on the analysis of differences in error handling across versions (see Section 4.3). This limitation also underscores the enhanced security of authenticated services, presenting a higher barrier to compromise.

Thirdly, while *VersionSeek* is theoretically applicable to closed-source or proprietary software systems, its practical effectiveness in such contexts may be constrained by the unavailability of detailed development information. Specifically, the lack of sufficient insights into these systems makes it more difficult to automatically analyze their programs and extract functional changes, thereby limiting the generation of effective probes. Expanding the probes in such cases may require additional manual efforts, such as reverse engineering or crowdsourced analysis.

Finally, some versions may lack suitable functional probes, preventing accurate version identification. Not all feature changes can be used to generate corresponding probes, such as dependency upgrades or performance enhancements. Additionally, certain adjacent versions with only patch-level differences (e.g., Elasticsearch 6.8.18-6.8.23) do not exhibit significant modifications beyond dependency upgrades, making them difficult to distinguish. Moreover, some probes require specific preconditions, for example, the Redis "CLUSTER INFO" probe needs a clustered server configuration, limiting their applicability to testing non-clustered servers.

7 Related Work

Side Channel. Freiling and Schinzel [40] proposed a storage side channel on web servers by comparing response differences across web applications, and used this to extract sensitive data such as existing usernames in public libraries and private images in real-world systems. In our work, our motivation example for distinguishing Elasticsearch versions is similar to this side channel.

Web application version identification. Prior works on web application version identification adopted two main approaches. Version string-based methods, such as Wappalyzer [30], use rule databases to infer software types and versions, and have been applied at scale to analyze web technology stacks [4, 5]. Feature-matching methods rely on structural features such as image file sizes, JavaScript function names [7], XPath structures, and file hashes [8] for website fingerprinting. Kondracki and Nikiforakis [9] summarized these techniques and proposed a sandbox framework for analyzing fingerprinting tools.

Fingerprinting applications. Website fingerprinting enables a local attack to determine which websites a user visits over an encrypted connection. Overdorf et al. [41] analyzed various website fingerprinting methods and evaluated 482 Tor onion services. Sirinam et al. [42] introduced a Tor attack using CNN, which can bypass some website fingerprinting defenses, while Guan et al. [43] proposed the Block Attention Profiling Model for identifying multiple websites in a global view. Browser fingerprinting leverages device and browser characteristics for tracking without cookies. Li and Cao [44] conducted a large-scale analysis of millions of browser fingerprints to study their dynamics, and Torok and Levy [45] proposed an entropy-based defense method.

8 Conclusion

We propose a novel version identification method based on functional changes and implemented a prototype framework, *VersionSeek*, on five well-known remote software systems with numerous vulnerabilities. *VersionSeek* achieved a significantly higher identification rate than existing version identification tools, with a 284% improvement on phpMyAdmin and a unique ability to identify Dubbo versions. Furthermore, we conducted a large-scale version identification experiment using *VersionSeek*, successfully identifying version information for 240,020 software instances, with 95.61% of the results achieving minor and patch-level identification. Analyzing the results, we discovered user preferences in version selection and highlighted the severe security vulnerabilities that could result from using outdated versions and failing to enable security features.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments, which improved the quality of the paper. This work was the National Natural Science Foundation of China (62302101, 62402114, U2436207). Haixin Duan's work was supported partly by the Taishan Scholars Program. Min Yang is the corresponding author, a faculty of Shanghai Institute of Intelligent Electronics & Systems and Engineering Research Center of Cyber Security Auditing and Monitoring, and Shanghai Collaborative Innovation Center of Intelligent Visual Computing, Ministry of Education, China.

Ethics Considerations

We introduce a novel approach to software version identification by analyzing functional changes, particularly in cases where existing scanning tools fail due to missing version banners or active security configurations. The purpose of this tool is to assist security researchers in identifying outdated or vulnerable systems that may otherwise be overlooked.

Recognizing the potential ethical risks associated with network scanning, we followed several design principles in tool implementation to operate in a way that minimizes disruption, avoids intrusion, and respects the operational integrity of the targeted systems.

Use of Official Functionalities. *VersionSeek*'s all probes are derived from standard functionalities publicly documented by the corresponding software vendors. The tool invokes their functionalities in their intended form, avoiding malformed or ambiguous inputs. Functionalities associated with bug fixes or known crash paths were excluded to reduce the risk of triggering server-side failures (see Section 2). Rather than attempting to exploit vulnerabilities, the approach focuses on benign differences in behavior across software versions.

Minimizing System Impact. To minimize the probability of impacting remote servers, the tool avoids any commands or methods that may alter server state or data. For instance, HTTP methods like POST, PUT, DELETE, and PATCH were excluded. For terminal-based protocols, commands that affect configuration, state, or storage (*e.g.,*, flush, shutdown, SET, DEL) were filtered out.

Avoiding Brute-force-like Behavior. Although some legitimate functionalities may involve login or directory traversal, the scanning logic should avoid repeated login attempts or recursive exploration. *VersionSeek* sent fewer than two login requests per server on average, well below the threshold of brute-force login attempts. Nevertheless, the tool does not exclude any form of username enumeration. Additionally, for HTTP services, only known, documented, and locally verified paths were used, and no blind directory enumeration was performed.

Polite Scanning. The tool is implemented to scan in a polite and resource-conscious manner. Firstly, as detailed in Section 3.3, an algorithm is developed to plan minimal probe sequences, with the goal of reducing the number of probes per scan. Secondly, the number and frequency of probes per host are limited strictly to prevent excessive scanning. Additionally, rather than sending all probes simultaneously, the tool schedules each subsequent probe based on the previous response, resulting in probes that are spaced out with timing intervals approximating normal user interactions. If a server refuses a connection, scanning ceases immediately for that host.

Data Use and Privacy. In our large-scale scanning experiments, scanned response data were used exclusively for the purpose of software version identification. No attempts were made to extract or correlate host-specific, personal, or confidential information. Data was handled in a manner that avoids unintentional disclosure or misuse.

Responsible Disclosure. We actively communicated with vendors and users to disclose our findings regarding the Elastic vulnerability and outdated or vulnerable servers detected on the Internet. First, we reported the potential username enumeration behavior, as demonstrated in our motivation example (Figure 1), to the Elastic team. In response, they clarified that merely confirming whether an email or username is associated with an account does not constitute a direct security impact or significantly facilitate further attacks.

For outdated or vulnerable servers identified in the real world, we conducted a responsible disclosure by retrieving WHOIS information associated with the server to extract administrator email contacts. We successfully identified 8,077 unique email addresses and successfully sent 4,983 responsible disclosure emails informing them of potential security risks. Over 86.9% of the responses were automated replies. 11 administrators expressed appreciation for our notification and indicated that they would attempt to update or patch the affected systems.

Open Science

We are committed to the open science policy and will make all relevant source code, datasets, and generated artifacts publicly available. However, response data and identification results derived from real-world servers will not be released, as public access to such information could potentially enable malicious exploitation or attacks against these servers.

We have made our code publicly available at https://doi.org/10.5281/zenodo.15576928, including modules for software deployment, functional probe generation, response processing, and version identification. The repository also contains ethically filtered and validated probes with their locally generated response outputs, as well as the source code and training data used for the comparative baseline.

References

[1] Nmap. (2024) nmap-service-probes. https://svn.nmap.org/nmap/nmap-service-probes.

[2] L. Bao, X. Xia, A. E. Hassan, and X. Yang, "V-szz: Automatic identification of version ranges affected by cve vulnerabilities," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 2352–2364.

[3] Elastic. Elasticsearch: The Official Distributed Search & Analytics Engine | Elastic — elastic.co. https://www.elastic.co/elasticsearch.

[4] F. Marquardt and L. Buhl, "Large scale monitoring of web application software distribution to measure threat response behavior," *Electronic Communications of the EASST*, vol. 80, 2021.

[5] Y. Shi, W. Yu, Y. Zhao, and Y. Jia, "A web application fingerprint recognition method based on machine learning." *CMES-Computer Modeling in Engineering & Sciences*, vol. 140, no. 1, 2024.

[6] GitHub - lokifer/BlindElephant: Getting BlindElephant into a working state, and updating the plugin files — github.com. https://github.com/lokifer/BlindElephant.

[7] C. Dresen, F. Ising, D. Poddebniak, T. Kappert, T. Holz, and S. Schinzel, "Corsica: Cross-origin web service identification," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 409–419.

[8] F. Marquardt and L. Buhl, "Déjà vu? client-side fingerprinting and version detection of web application software," in *2021 IEEE 46th Conference on Local Computer Networks (LCN)*. IEEE, 2021, pp. 81–89.

[9] B. Kondracki and N. Nikiforakis, "Smudged fingerprints: Characterizing and improving the performance of web application fingerprinting," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4625–4640.

[10] Apache. (2024) Apache dubbo. https://cn.dubbo.apache.org/.

[11] Redis. (2024) Redis github. https://github.com/redis/redis.

[12] Joomla. (2024) Joomla github. https://github.com/joomla/joomla-cms.

[13] phpMyAdmin Project, "phpmyadmin – a web interface for mysql and mariadb," https://www.phpmyadmin.net/, 2025.

[14] Shodan. (2013) Shodan.io. https://www.shodan.io/.

[15] FOFA. (2024) Fofa search engine. https://fofa.info/.

[16] G. Lyon. (2024) Nmap: the network mapper - free security scanner. https://nmap.org/.

[17] (April 23, 2024) Usage statistics and market shares of content management systems. https://w3techs.com/technologies/overview/content_management.

[18] (June 23, 2024) Apache: Dubbo Security Vulnerabilities. https://cvedb.shodan.io/dashboard/vulnerabilities?product=dubbo.

[19] (June 5, 2024) Elasticsearch Security Vulnerabilities. https://vulners.com/search?query=elasticsearch.

[20] (June 5, 2024) Redis Security Vulnerabilities. https://vulners.com/search?query=redis.

[21] (June 5, 2024) phpmyadmin Security Vulnerabilities. https://vulners.com/search?query=phpmyadmin.

[22] Metasploit. Metasploit the world's most used penetration testing framework. https://www.metasploit.com/.

[23] (2024) Whatweb. https://github.com/urbanadventurer/WhatWeb.

[24] ServerMask : The Official Microsoft IIS Site. https://iis-umbraco.azurewebsites.net/downloads/community/2009/01/servermask.

[25] (2024) Semantic Versioning. https://semver.org/.

[26] I. GitHub. (2022) Github rest api. https://docs.github.com/en/rest.

[27] (2024) Jubbo showcase. https://showcase.joomla.org/.

[28] Hobbit. (2024) Netcat introduction. https://nmap.org/ncat/.

[29] (2024) Llm leaderboard. https://artificialanalysis.ai/leaderboards/models.

[30] E. Alias. (2024) Find out what websites are built with - wappalyzer. https://www.wappalyzer.com/.

[31] T. Pascal Wichmann, Martin Müller. (2024) Versioninferrer Github. https://github.com/wichmannpas/VersionInferrer.

[32] Zgrab. (2024) Zgrab 2.0. https://github.com/zmap/zgrab2.

[33] Nmap. (2024) Nmap scripting engine. https://www.metasploit.com/.

[34] M. Wu, G. Hong, J. Chen, Q. Liu, S. Tang, Y. Li, B. Liu, H. Duan, and M. Yang, "Revealing the black box of device search engine: Scanning assets, strategies, and ethical consideration," in *Proceedings of the Network and Distributed System Security (NDSS) Symposium 2025*, ser. NDSS '25, 2025. [Online]. Available: https://dx.doi.org/10.14722/ndss.2025.241924

[35] NIST. (2022) cvedetails.com. https://www.cvedetails.com/.

[36] NIST. (2023) Cve-2023-31418. https://nvd.nist.gov/vuln/detail/cve-2023-31418.

[37] (2024) Security for Elasticsearch. https://www.elastic.co/blog/security-for-elasticsearch-is-now-free/.

[38] NIST. (2023) Cve-2019-19846. https://nvd.nist.gov/vuln/detail/cve-2019-19846.

[39] NIST. (2022) Cve-2022-23797. https://nvd.nist.gov/vuln/detail/cve-2022-23797.

[40] F. C. Freiling and S. Schinzel, "Detecting hidden storage side channel vulnerabilities in networked applications," in *IFIP International Information Security Conference*. Springer, 2011, pp. 41–55.

[41] R. Overdorf, M. Juarez, G. Acar, R. Greenstadt, and C. Diaz, "How unique is your .onion? an analysis of the fingerprintability of tor onion services," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2021–2036. [Online]. Available: https://doi.org/10.1145/3133956.3134005

[42] P. Sirinam, M. Imani, M. Juarez, and M. Wright, "Deep fingerprinting: Undermining website fingerprinting defenses with deep learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1928–1943. [Online]. Available: https://doi.org/10.1145/3243734.3243768

[43] Z. Guan, G. Xiong, G. Gou, Z. Li, M. Cui, and C. Liu, "Bapm: Block attention profiling model for multi-tab website fingerprinting attacks on tor," in *Proceedings of the 37th Annual Computer Security Applications Conference*, ser. ACSAC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 248–259. [Online]. Available: https://doi.org/10.1145/3485832.3485891

[44] S. Li and Y. Cao, "Who touched my browser fingerprint? a large-scale measurement study and classification of fingerprint dynamics," in *Proceedings of the ACM Internet Measurement Conference*, ser. IMC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 370–385. [Online]. Available: https://doi.org/10.1145/3419394.3423614

[45] R. Torok and A. Levy, "Only pay for what you leak: Leveraging sandboxes for a minimally invasive browser fingerprinting defense," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1023–1040.

[46] (2024) Elasticsearch release notes. https://www.elastic.co/guide/en/elasticsearch/reference/8.15/es-release-notes.html.

[47] (2024) Redis release notes. https://github.com/redis/redis/tags.

[48] (2024) Dubbo release notes. https://github.com/apache/dubbo/releases/tag/dubbo-3.2.15.

[49] (2024) Joomla release notes. https://github.com/joomla/joomla-cms/releases.

[50] (2025) phpmyadmin news. https://www.phpmyadmin.net/news/.

[51] (2024) Elasticsearch User's Guide. https://www.elastic.co/guide/en/elasticsearch/reference/current/docs.html.

[52] (2024) Redis User's Guide. https://redis.io/docs/latest/index.xml.

[53] (2024) Dubbo User's Guide. https://cn.dubbo.apache.org/en/overview/what/.

[54] (2024) Joomla User's Guide. https://docs.joomla.org/Main_Page.

[55] (2025) phpmyadmin User's Guide. https://docs.phpmyadmin.net/en/latest/.

[56] (2024) Beautifulsoup. https://www.crummy.com/software/BeautifulSoup/.

[57] H. Face. (2024) Qwen2.5-32b-instruct. https://huggingface.co/Qwen/Qwen2.5-32B-Instruct.

[58] qwen. (2024) Qwen-agent. https://github.com/QwenLM/Qwen-Agentl.

A Probe Planning Algorithm

Here we provide our algorithm of minimal probe sequence selection (Algorithm 2) and Decision tree generation (Algorithm 3).

Algorithm 2: Minimal Probe Sequence Selection

```
Data: Version set $V$, available probes $P$.
Result: Minimal probe sequence $P_{min}$ distinguish $V$.
Function minimalProbes($V, P$):
    $optResults \leftarrow \{\}, P_{min} \leftarrow [\,]$
    foreach $v \in V$ do
        $P_{temp}, V_{notDistinguished} \leftarrow$ selectGreedy$(v, P)$
        $optResults[v] \leftarrow V_{notDistinguished}$
        $P_{min}$.extend$(P_{temp})$
    end
    $P_{remain} \leftarrow P_{min}$
    foreach $p \in P_{min}$ do
        $P_{remain}$.remove$(p)$
        $isRedundant \leftarrow$ True
        foreach $v \in V$ do
            $P_{temp}, V_{notDistinguished} \leftarrow$ selectGreedy$(v, P_{remain})$
            if $V_{notDistinguished} \neq optResults[v]$ then
                $isRedundant \leftarrow$ False
                break
            end
        end
        if $isRedundant$ then
            $P_{min} \leftarrow P_{remain}$
            $P_{min} \leftarrow$ minimalProbes($V, P_{min}$)
            return $P_{min}$
        end
        $P_{remain}$.append$(p)$
    end
    return $P_{min}$
```

Algorithm 3: Decision Tree Construction

```
Input: Current probe $p_{cur}$, version set $V$, minimal probe sequence $P$, path from root node to current node $path$
Output: Decision tree node $node$
Function BuildTree($p_{cur}$, $V$, $P_{used}$, $path$):
    $node \leftarrow$ CreateNode$(p_{cur}, V, path)$, $R_{cur} \leftarrow$ response set of $p_{cur}$
    foreach $r_j \in R_{cur}$ do
        $A \leftarrow f(p_{cur}, r_j)$; $V_{new} \leftarrow V \cap A$;
        $branch\_name \leftarrow p_{cur}\_resp_j$
        if $V_{new} = \emptyset$ or $V_{new} = V$ then
            continue
        end
        else if $|V_{new}| = 1$ then
            $path$.append$(branch\_name)$
            $childNode \leftarrow$ CreateLeafNode$(p_{cur}, V_{new}, path)$
            $node$.addChild$(childNode)$
        end
        else
            Select a probe $p_k \in P$ that can split $V_{new}$ (if such a probe exists);
            if no such probe exists then
                $path$.append$(branch\_name)$
                $childNode \leftarrow$ CreateLeafNode$(p_{cur}, V_{new}, path)$
                $node$.addChild$(childNode)$
            end
            else
                $P$.remove$(p_k)$
                $path$.append$(branch\_name)$
                $childNode \leftarrow$ BuildTree($P$, $V_{new}$, $path$)
                $node$.addChild$(childNode)$
                $P$.append$(p_k)$
            end
        end
    end
    return $node$
```

B Implementation

This section implements our functionality-based version identification method on five open-source software for local experiments: ElasticSearch, Dubbo, Redis, Joomla and phpMyAdmin. We use them to evaluate the effectiveness of our method and provide data support for large-scale identification of real-world software versions.

We conducted experiments on a server with a 13th Gen Intel® Core™ i7-13700 processor (24 cores), 32GB of memory, and a 1TB SSD, running the Ubuntu 23.04 operating system.

Version Selection. As outlined in Section 4, we selected ElasticSearch, Dubbo, Redis, Joomla and phpMyAdmin to demonstrate the capabilities of our version identification tool.

For each software, we selected stable releases from the past decade (2014-2024). We filtered out pre-release and development versions, as these are typically not used in production environments. Additionally, while our deployment strategy aims to cover all versions, we failed to include some versions due to obsolescence or the lack of maintenance for their dependent libraries. Table 2 shows the softwares and corresponding versions we covered.

Deployment. We deployed the software using containerized services based on the official source code. Specifically, we accomplished this by utilizing the original images provided by the official sources or by building images based on the official test examples. If an official Docker image is available, we change the image versions to get rapid deployment of different versions. If not, we compiled and ran the code locally based on the official examples, packaging it into a Docker image according to the provided instructions. Additionally, we used a docker-compose.yml file to manage multiple docker images and generate rapid deployment commands for each version.

We try our best to minimize discrepancies between local experiments and real-world environments. We did not introduce any user-defined content, such as new data or plugins.

Notably, some software systems provide security mechanisms, such as authentication requirements, restrictions on external IP access, or blocking external IPs, to enhance safety. These mechanisms may render some probes accessible in local environments, intercepted, or ineffective in real-world scenarios. To cope with this situation, we conducted experiments under different configurations based on the security mechanisms provided by the software, as summarized in Table 9.

Documentation Collection. We implemented a web crawler in Python to collect functional features and contextual information from release notes [46, 47, 48, 49, 50] and Users' Guides [51, 52, 53, 54, 55], which are typically web documents. We used `BeatuifulSoup4` [56] to parse the release documents, extracted the description of each functional feature and its corresponding Pull Request ID, and then called the GitHub API to retrieve descriptions from the pull request webpage. Additionally, we excluded any functional features for which context retrieval failed. The results of the software documentation collection are summarized in Table 2. We extracted 7,963 functional changes from the release notes of 684 different versions across the five software.

Probe Generation. We used the open-source Qwen2.5-32B-Instruct model [57] and the accompanying Qwen-Agent Python framework [58]. We evaluated different models in Section 4.2. The Qwen-Agent framework provides a "code_interpreter" Agent, which allows Qwen to execute code and retrieve the results. Building on this, we developed the functional probe generation system and conducted large-scale probe generation.

However, LLMs do not generate valid probes for every functional feature, so we filter out responses lacking actionable instructions, such as "There is no specific command provided in the given context to trigger..". Additionally, although we give LLM the example probe format in the prompt, LLM may still make mistakes. For example, some probes only provide the request method and path (e.g., GET /_search) but lack essential parameters such as host and port, while some probes contain redundant explanations, such as "the specific command would be". To address this, we processed raw responses using a regex-based method, retaining only commands and configuring appropriate parameters to match the format of the provided examples.

Response Processing. We implemented an automated Python-based framework to handle each version of the software's workflow sequentially, including deployment, probe execution, response collection, and uninstallation, while monitoring the status at each stage. After obtaining the responses, we mask environment variables unrelated to version-specific functional features, such as resource usage, IP addresses, and network transmission rates.

For each probe, we grouped versions with identical responses into the same category to determine the version classification and retained only the probes that can categorize versions into at least two distinct classes, dynamically generated version identification decision trees for version identification.

Version Identification. Based on the effective probes collected from local experiments and a classification function that maps response types to corresponding versions for each probe, we implemented a Python-based version identification scanner capable of identifying versions on real-world servers.

Specifically, *VersionSeek* takes as input the target server address and the corresponding software type. It first uses "nc -z" to check whether the relevant port is open. For servers with open ports, *VersionSeek* invokes the appropriate set of probes and the corresponding classification function based on the software type. Following the procedure outlined in Algorithm 1, we implemented a Python class to simulate the decision tree structure, which is serialized and deserialized using the JSON format for storage and reuse. The responses are normalized based on locally collected rules and then processed using Python's re library to match and replace noise. Based on the decision tree, *VersionSeek* maps the matched response to a set of candidate versions.

If a probe fails to match, *VersionSeek* re-plans the probing schedule using remaining unused probes and the current candidate version set. This process continues until a final, indistinguishable version set is reached. In case of conflicting results, a majority voting algorithm selects the version supported by the most matching probes.

C Evaluation

C.1 Probe Effectiveness

Table 9 shows the result of testing the generated probes under different security configurations, retaining only those that produced distinguishing responses across versions.

C.2 Scanning Efficiency

Table 7 shows the average time for probe generation, scheduling, and execution for different services and the average number of probes and scanning time per server.

Software Service	Average Time for Generate One Probe	Average Time for Probe Scheduling	Average Time for One Probe Execution	Average Number of Probes Per Server Scan	Average Time to Scan One Server
Elasticsearch	55.706s	4.425s	1.021s	5.800	7.400s
Dubbo	56.041s	0.001s	2.876s	4.867	14.611s
Redis	60.710s	2.372s	1.287s	2.633	7.440s
Joomla	69.556s	3.824s	1.458s	6.233	9.130s
phpMyAdmin	68.634s	0.513s	0.574s	9.416	7.969s

Table 7: Average time for probe generation, scheduling, and execution for different services and the average number of probes and scanning time per server.

Software	*VersionSeek*		Nmap		Metasploit		Blindelephant		WhatWeb	
	obf	hid	obf	hid	obf	hid	obf	hid	obf	hid
Elasticsearch	100%	100%	0%	0%	0%	0%	-	-	-	-
Redis	100%	100%	0%	0%	0%	0%	-	-	-	-
Joomla	100%	100%	-	-	0%	0%	29%	29%	0%	0%
phpMyAdmin	100%	100%	-	-	-	-	100%	100%	0%	0%

Table 8: Version Identification Performance of Different Tools under Obfuscated(obf) and Hiding(hid) Scenarios.

Software	Security Configurations	# of Valid Probes	Length of Minimal Probe Sequence
Elasticsearch	with auth	544	8
	no auth	488	23
Redis	with auth	98	10
	no auth	132	12
	deny external access	1	1
Dubbo	-	282	4
Joomla	-	114	5
phpMyAdmin	-	176	9

Table 9: The effectiveness of probes under different security configurations for software.

C.3 Robustness of Version Identification

Table 8 shows the version identification performance of different tools under obfuscated (obf) and hiding (hid) scenarios.