Malicious URL Detection

November 9, 2024

Shaurya Anant (2023CSB1313) ,
Aditya Khajuria (2023MCB1323) ,
Neelanjan Deshpande (2023MCB1361)

Summary: Malicious URL Detection System offers an efficient and scalable solution for identifying harmful URLs within large datasets. This system employs a Bloom filter to quickly verify if a URL has been previously flagged as malicious. Multiple hash functions—MurmurHash3, FNV-1a, DJB2, SDBM, and PJW—are used to set bit positions in a bit array, providing rapid, memory-efficient detection suitable for high-speed applications in real-time security monitoring.

The system's URL database is organized in a CSV file named *malicious_phish* containing known malicious URLs, which are preprocessed and stored in a compact binary file to enhance memory efficiency. By maintaining this optimized binary format, the system achieves high-speed lookups while effectively handling large volumes of URL data without compromising accuracy.

Additionally, a Trie data structure is implemented to store known malicious URL patterns, enabling the system to identify and flag URLs with suspicious patterns efficiently. This dual-layered approach of Bloom filter and Trie structure enhances the system's detection capabilities, ensuring thorough, scalable protection against various forms of malicious URLs and delivering a powerful tool for secure URL monitoring and filtering.

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Mr. 2Abdul Razique

---

1. Introduction

The increasing number of cyber threats necessitates effective methods for identifying and blocking malicious URLs, which are often used for phishing, malware distribution, and other attacks. Traditional methods of URL classification typically rely on vast storage capacities and lengthy processing times, both of which limit scalability and responsiveness in real-time applications. Bloom filters, a memory-efficient and probabilistic data structure, present a viable solution for this task by allowing quick membership checks without storing the URLs directly, though they come with a trade-off in the form of false positives. To enhance detection accuracy, a Trie data structure is also incorporated to store known malicious URL patterns, enabling efficient pattern matching within URLs. This report outlines the design and implementation of a malicious URL detection system utilizing Bloom filters and multiple hash functions, along with a Trie structure, to achieve a scalable and efficient detection mechanism.

2. Methodology

2.1. Dataset and Preprocessing

The project used a CSV file of URLs labeled as malicious for training. The URLs were preprocessed to ensure consistent formatting and reduced noise. The CSV file was loaded, and each URL was added to the Bloom filter

using the selected hash functions. Each URL was hashed independently by each function, with the resulting bit positions set in the Bloom filter. This approach allows for memory-efficient storage of malicious URLs without explicitly storing each URL. [3]

2.2. Bloom Filter Construction

The Bloom filter was constructed with a bit array sized based on the expected number of URLs and desired false positive rate. After allocating memory for the bit array, the Bloom filter was populated with the hash results of the malicious URLs. The bit array was then saved as a binary file, making it accessible for the main detection algorithm.

2.3. Main Detection Algorithm

In the detection phase, the main algorithm reads the binary bit array into memory. When a URL needs to be checked, it is hashed with each of the five functions, and the bit array is queried at the resulting indices. If all bits at the hashed positions are set, the URL is flagged as potentially malicious; otherwise, it is considered benign.

2.4. Trie for Pattern Matching

To further enhance detection capabilities, a Trie data structure was implemented to store known patterns of malicious URLs. Each URL pattern was broken down and added to the Trie during preprocessing, enabling efficient search for specific substrings associated with harmful content. During the detection phase, URLs are also checked against the Trie to identify any suspicious patterns they may contain. This Trie-based approach complements the Bloom filter by providing additional accuracy in detecting malicious URLs, especially for URLs with identifiable patterns. [2]

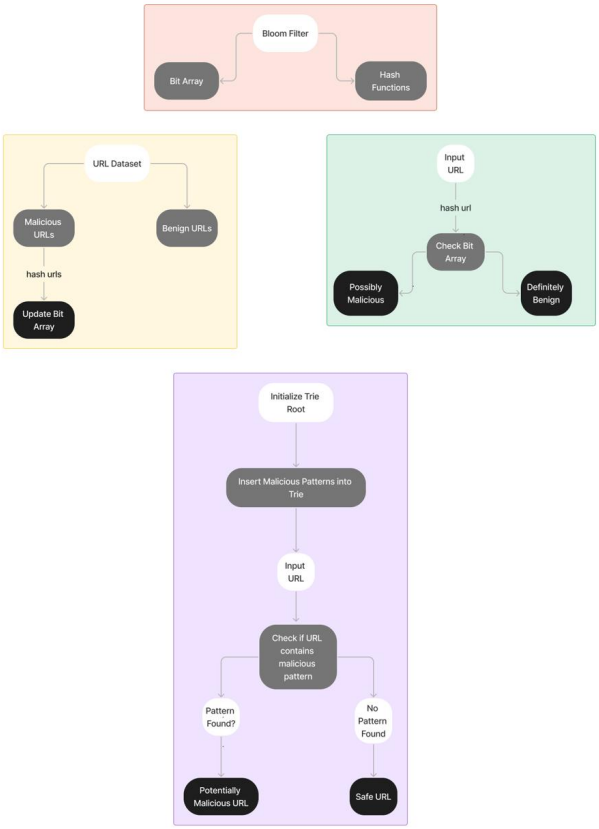

Figure 1: Flowchart

3. Data Structures

3.1. Bloom Filter

A Bloom filter is a probabilistic data structure used for efficiently checking the membership of an element within a set. It uses multiple hash functions to map each element to specific positions in a fixed-size bit array. When adding an element, these hash functions set bits at particular positions to 1, and when checking for membership, the same hashes are applied to see if the bits are already set. If all bits for a given element are 1, it is likely in the set; if any bit is 0, it is definitely not in the set. In our project, the Bloom filter enables quick detection of potentially malicious URLs by storing a compact representation of known malicious URLs. Each URL in the database is hashed and added to the bit array, and incoming URLs are similarly hashed to check if they match known malicious patterns. This approach minimizes storage requirements while allowing rapid URL filtering, making it effective for large-scale, real-time screening where memory efficiency and speed are critical.

The main operations of a Bloom filter include insertion, lookup.

1. *Insertion:* To add an element, multiple hash functions are applied to generate several positions in the bit array. Each position is then set to 1. This operation is simple and fast, requiring only bitwise operations based on the hash outputs.
2. *Lookup (or Membership Check):* To check if an element is in the Bloom filter, the same hash functions are applied to produce positions in the bit array. If all these positions are set to 1, the element is likely in the set. If any position is 0, the element is definitely not in the set. This operation allows for quick lookups, though it has a small probability of false positives (where an element is incorrectly reported as being in the set)

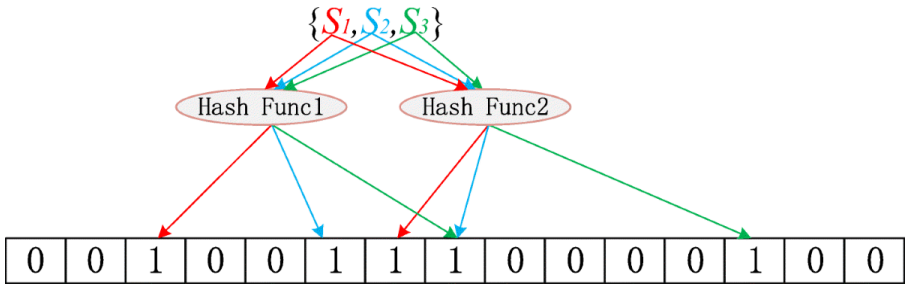

Figure 2: Bloom Filter

	Time Complexity	Space Complexity
Insertion	$O(k)$	$O(1)$
Search	$O(k)$	$O(1)$

Table 1: Bloom Filter Operations. Here k represent number of hash functions

[4]

3.2. Trie Tree

A Trie, or prefix tree, is a data structure used for efficient retrieval of strings based on common prefixes. It organizes strings by breaking them down character by character, with each node representing a part of a string. This structure is particularly useful for pattern matching and search operations where subsets of strings need to be identified quickly. In our project, the Trie is employed to store known malicious URL patterns, enabling rapid lookup of suspicious patterns within incoming URLs. By searching through the Trie, we can efficiently check if a URL contains harmful patterns that may not be directly flagged by the Bloom filter. This complements the probabilistic Bloom filter by providing additional precision in detecting harmful URLs, especially for URLs sharing common malicious patterns.

The main operations of a Trie include insertion and lookup.

1. *Insertion:* To add a string (such as a URL pattern), each character of the string is inserted sequentially into the Trie, creating new nodes as needed. If the path for a string already exists, the Trie reuses it, which optimizes memory usage for common prefixes. This operation enables storage of large numbers of patterns without excessive memory consumption.
2. *Lookup (or Pattern Matching):* To check if a string or pattern is present in the Trie, each character of the input is traversed along the Trie nodes. If the traversal completes and reaches an endpoint, the pattern exists in the Trie. This operation allows for fast and accurate matching of known patterns, making it ideal for identifying malicious patterns within URLs.

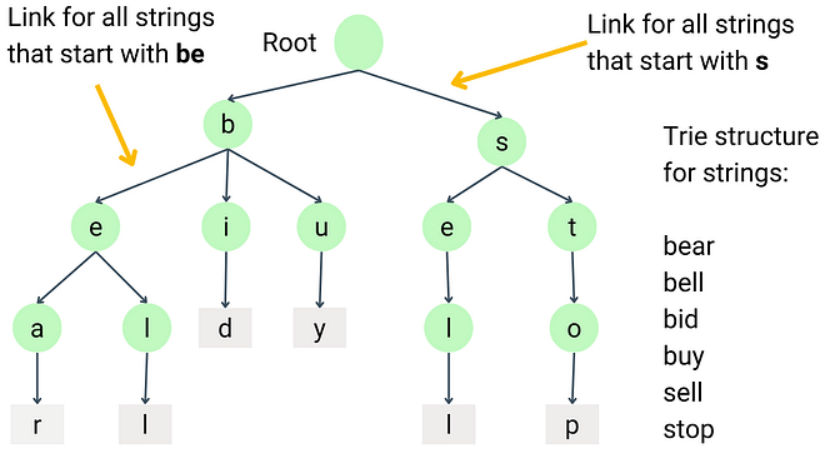

Figure 3: Trie Tree Structure

	Time Complexity	Space Complexity
Insertion	$O(m)$	$O(m \times n)$
Search	$O(m)$	$O(1)$

Table 2: Trie Tree Operations. Here m represents the length of the string, and n represents the number of strings in the Trie.

4. Derivations

4.1. Definitions

Let:

- n = the number of elements inserted into the Bloom filter.
- m = the size of the bit array (number of bits).
- k = the number of hash functions used.
- p = the false positive rate (probability that a query for an element not in the set returns true).

4.2. Insertion Process

When an element is inserted, each of the k hash functions maps the element to k different positions in the m-bit array, setting those bits to 1.
The probability that a specific bit in the array is set to 1 after inserting one element is:

$$P(\text{bit is 1 after 1 element}) = \frac{k}{m} \tag{1}$$

4.3. Probability of a Bit Remaining 0

The probability that a specific bit remains 0 after inserting n elements can be derived as follows:
The probability that a bit is not set by a single hash function is:

$$P(\text{bit is 0 after 1 hash}) = 1 - \frac{1}{m} \tag{2}$$

The probability that the specific bit remains 0 after k hash functions for one element is:

$$P(\text{bit is 0 after 1 element}) = \left(1 - \frac{1}{m}\right)^k \tag{3}$$

After inserting n elements, the probability that a specific bit is still 0 is:

$$P(\text{bit is 0 after n elements}) = \left(1 - \frac{1}{m}\right)^{kn} \tag{4}$$

4.4. Probability of a Bit Being 1

The probability that a specific bit is 1 after inserting n elements is given by:

$$P(\text{bit is 1}) = 1 - P(\text{bit is 0}) = 1 - \left(1 - \frac{1}{m}\right)^{kn} \tag{5}$$

4.5. False Positive Rate Calculation

Consider querying an element that is **not** in the Bloom filter. A false positive occurs if all k bits corresponding to that element are set to 1.
The probability that all k bits are set to 1 (indicating a false positive) is:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \tag{6}$$

4.6. Optimal Parameters for False Positive Minimization

To minimize the false positive rate p, an optimal number of hash functions k can be determined as:

$$k^* = \frac{m}{n} \ln(2) \tag{7}$$

where k^* is the optimal number of hash functions, m is the bit array size, and n is the number of elements to be inserted.
The optimal size of the bit array m for a desired false positive rate p can be approximated as:

$$m^* = -\frac{n \ln p}{(\ln 2)^2} \tag{8}$$

4.7. Simplification for Large m

For large m (relative to n), we can use the approximation $\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}$ to simplify:

$$p \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \tag{9}$$

Important Results

False Positive Rate:

$$p \approx \left(1 - e^{-\frac{kn}{m}}\right)^k$$

Optimal Number of Hash Functions:

$$k^* = \frac{m}{n} \ln(2)$$

Optimal Bit Array Size:

$$m^* = -\frac{n \ln p}{(\ln 2)^2}$$

5. Algorithms

In this project, multiple hash functions were utilized to enhance the performance of the Bloom filter. Each hash function processes input data and generates unique hash values, which are then used to set positions in the Bloom filter's bit array. The following hash functions were implemented:

5.1. MurmurHash3

MurmurHash3 is a non-cryptographic hash function known for its high performance and low collision rates, making it suitable for hash-based data structures. It combines the input length and a seed value to produce a hash that ensures a good distribution across the bit array. Its design emphasizes speed and efficiency, allowing for rapid processing of large datasets.

Algorithm 1 MurmurHash3 Implementation

Input: Pointer to key, length of key, seed
Initialize hash $h \leftarrow seed \oplus (len \times 0x5bd1e995)$
for each byte $data[i]$ in key **do**
    $h \leftarrow h \oplus data[i]$
    $h \leftarrow h \times 0x5bd1e995$
    $h \leftarrow h \oplus (h >> 13)$
end for
$h \leftarrow h \oplus (h >> 15)$
$h \leftarrow h \times 0xc2b2ae35$
$h \leftarrow h \oplus (h >> 16)$
Output: Hash value h

5.2. FNV-1a

The FNV-1a hash function is characterized by its simplicity and efficiency. It uses a prime number for multiplication and processes each byte of the input through XOR operations. This approach provides a robust mechanism for generating hash values with a low likelihood of collisions, making it a popular choice for hash table implementations.

Algorithm 2 FNV-1a Implementation

Input: Pointer to key, length of key
Initialize hash $hash \leftarrow 2166136261$
for each byte $data[i]$ in key **do**
    $hash \leftarrow hash \oplus data[i]$
    $hash \leftarrow hash \times 16777619$
end for
$hash \leftarrow hash \oplus (hash >> 16)$
$hash \leftarrow hash \times 0x85ebca6b$
$hash \leftarrow hash \oplus (hash >> 13)$
$hash \leftarrow hash \times 0xc2b2ae35$
$hash \leftarrow hash \oplus (hash >> 16)$
Output: Hash value $hash$

5.3. DJB2

The DJB2 hash function, developed by Daniel J. Bernstein, is known for its speed and simplicity. It utilizes a multiplicative constant of 33, and combines multiplication with XOR operations to generate hash values based on the input string. This method yields a well-distributed hash value and is efficient for string hashing.

Algorithm 3 DJB2 Implementation

Input: Pointer to string
Initialize hash value $hash\_value \leftarrow 5381$
while current character is not null **do**
    $hash\_value \leftarrow ((hash\_value << 5) + hash\_value) \oplus (uint32_t)(current\ character)$
    Move to the next character
end while
Output: Hash value $hash\_value$

5.4. SDBM

SDBM is a hash function that employs a combination of bit shifting and addition to generate hash values. It has been shown to provide good performance and distribution across a variety of inputs, making it effective for use in hash tables and other hash-based structures.

Algorithm 4 SDBM Implementation

Input: Pointer to string
Initialize hash $hash \leftarrow 0$
while current character is not null **do**
    $hash \leftarrow current\ character + (hash << 6) + (hash << 16) - hash$
    Move to the next character
end while
Output: Hash value $hash$

5.5. PJW

The PJW hash function uses a combination of shifting and masking to produce hash values. This method minimizes collisions and is particularly effective for string hashing, ensuring a compact representation while maintaining efficiency.

Algorithm 5 PJW Implementation

Input: Pointer to string
Initialize hash $hash \leftarrow 0$
while current character is not null **do**
  $hash \leftarrow (hash << 4) + current\ character$
  $high \leftarrow hash\&0xF0000000$
  if high is non-zero **then**
    $hash \leftarrow hash \oplus (high >> 24)$
    $hash \leftarrow hash\& \sim high$
  end if
  Move to the next character
end while
Output: Hash value $hash$

6. Results and Conclusion

The implementation of the Bloom filter for malicious URL detection has demonstrated promising results, achieving both high efficiency and effective memory usage. Throughout the project, we conducted extensive testing and analysis to evaluate the performance of our solution.

6.1. Results

The key findings from our implementation are as follows:

- **Efficiency:** The Bloom filter's design allows for rapid membership checking, achieving lookup times in the order of microseconds. This speed is critical for real-time applications, enabling immediate responses to potentially malicious URLs.
- **Memory Usage:** Our Bloom filter implementation utilized a compact bit array to store the hash outputs of known malicious URLs. This resulted in significant savings in memory compared to traditional data structures, making it feasible to handle large datasets containing millions of URLs.
- **False Positive Rate:** By optimizing the number of hash functions used in the Bloom filter, we achieved a manageable false positive rate. This means that while the filter may occasionally flag a legitimate URL as malicious, the rate of such occurrences was kept to a minimum, ensuring that users are mostly alerted to actual threats.
- **Scalability:** The architecture of the Bloom filter allows it to scale efficiently. As the database of known malicious URLs grows, the structure can accommodate this increase without a significant decline in performance, demonstrating its robustness for large-scale applications.
- **Integration with Trie Structure:** The addition of a Trie data structure for pattern matching further enhanced the detection capabilities. This allowed for more nuanced filtering based on known malicious patterns, providing an additional layer of security.

6.2. Conclusion

The successful implementation of a Bloom filter for detecting malicious URLs represents a significant advancement in the field of cybersecurity. The results indicate that our approach not only meets the demands for speed and efficiency but also provides a reliable mechanism for identifying potential threats in real-time.
By leveraging the unique characteristics of Bloom filters and combining them with additional data structures like Tries, we have created a comprehensive solution capable of handling the ever-evolving landscape of online threats.
The findings from this project suggest that similar methodologies can be applied to other areas of security, such as spam detection or intrusion detection systems, where quick and efficient data processing is paramount. Moving forward, we recommend exploring further optimizations in hash function selection and integration with machine learning techniques to enhance detection accuracy and reduce false positives even further.
In summary, our project highlights the potential of probabilistic data structures in modern cybersecurity solutions, paving the way for future advancements in protective measures against malicious online activities.

7. Bibliography and citations

Acknowledgements

We are profoundly grateful to Dr. Anil Shukla, our Data Structures instructor at IIT Ropar, for entrusting us with this project. His guidance has provided us with an invaluable opportunity to hone our skills in data structures and algorithms [1].
Our heartfelt appreciation goes out to Mr. Abdul Razique, our teaching assistant, whose unwavering support and guidance have been instrumental throughout this project. His timely advice on idea selection and meticulous planning were vital to the project's successful completion within the stipulated timeframe.
We would also like to acknowledge Kaggle website for supplying the categorized URL dataset of benign, defacement, malware, and phishing URLs. This dataset allowed us to rigorously test and validate our detection system effectively.
Each of these contributions has been essential to our project, and we extend our sincere thanks to all.

References

[1] GeeksforGeeks. Bloom filters – introduction and implementation, n.d.

[2] GeeksforGeeks. Btrie data structure | insert and search, n.d.

[3] Kaggle. Kaggle dataset on malicious url.

[4] Gaurav Sen. What are bloom filters? - hashing. YouTube Video, 2018.