A NOVEL AND PRECISE FALSE POSITIVE PROBABILITY COMPUTATION FOR BLOOM FILTERS IMPLEMENTED WITH UNIVERSAL HASH FUNCTIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

FURKAN KOLTUK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

AUGUST 2022

Approval of the thesis:

A NOVEL AND PRECISE FALSE POSITIVE PROBABILITY COMPUTATION FOR BLOOM FILTERS IMPLEMENTED WITH UNIVERSAL HASH FUNCTIONS

submitted by **FURKAN KOLTUK** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. İlkay Ulusoy
Head of Department, **Electrical and Electronics Engineering** _____

Prof. Dr. Ece Güran Schmidt
Supervisor, **Electrical and Electronics Engineering, METU** _____

Examining Committee Members:

Prof. Dr. İlkay Ulusoy
Electrical and Electronics Engineering, METU _____

Prof. Dr. Ece Güran Schmidt
Electrical and Electronics Engineering, METU _____

Assist. Prof. Dr. Serkan Sarıtaş
Electrical and Electronics Engineering, METU _____

Assist. Prof. Dr. Ahmed Hareedy
Electrical and Electronics Engineering, METU _____

Prof. Dr. Nail Akar
Electrical and Electronics Engineering, Bilkent University _____

Date: 22.08.2022

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Furkan Koltuk

Signature :

ABSTRACT

A NOVEL AND PRECISE FALSE POSITIVE PROBABILITY COMPUTATION FOR BLOOM FILTERS IMPLEMENTED WITH UNIVERSAL HASH FUNCTIONS

Koltuk, Furkan
M.S., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Ece Güran Schmidt

August 2022, 67 pages

Bloom Filters (BF) are multiple-hashing data structures that are widely used in membership testing applications. The many-to-one nature of the BF hashing results in false positive outcomes which have to be further processed at a performance cost. The computation of the false positive probability of BFs is carried out under the assumption of uniform and independent hash functions. To the best of our knowledge, all previous work in the literature assume that the hash functions are uniform and independent without verifying if that is the case in reality.

This thesis focuses on the hash function uniformity and independence for BFs with universal $H3$ functions. To this end, we propose a formal framework for defining and quantifying the uniformity and independence for H_3 hash functions in BFs. Furthermore, we define a formal description of the many-to-one outcomes of H_3 hash functions. We then use this framework to precisely compute the false positive probability for BFs with $H3$ hash functions which might not be necessarily uniform or independent. We verify our precise false positive expression with a hardware test bed

that can execute $2 \cdot 10^8$ membership queries per second. We perform structured tests to evaluate the effect of losing uniformity and independence at different levels among the H_3 hash functions. Furthermore, we evaluate the results of our expression for a range of parameters.

Keywords: Bloom Filter, Universal Hash Functions, False Positive Probability, Hash Function Uniformity and Independence

ÖZ

EVRENSEL ÖZET FONKSİYONLARI İLE GERÇEKLEŞTİRİLMİŞ BLOOM FILTRELERİ İÇİN YENİ VE HASSAS BİR YANLIŞ POZİTİF OLASILIĞI HESAPLAMASI

Koltuk, Furkan
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Ece Güran Schmidt

Ağustos 2022 , 67 sayfa

Bloom Filtreleri (BF), üyelik testi uygulamalarında yaygın olarak kullanılan çoklu özetleme foksiyonu içeren veri yapılarıdır. Bloom Filterelerindeki özet fonksiyonlarının çoğuldan tekile yapısı yanlış pozitif çıktılara ve neticede performans kayıplarına sebep olur. Literatürdeki Bloom Filtresi yanlış pozitif olasılığı hesaplamaları tekdüze ve bağımsız özet fonksiyonları varsayar. Bildiğimiz kadarıyla, literatürdeki tüm çalışmalar herhangi bir doğrulama sağlamaksızın özet fonksiyonlarının tekdüze ve bağımsız olduğunu varsaymaktadır.

Bu tez çalışması, $H3$ özet fonksiyonuna sahip BF'lerin tekdüzeliğine ve bağımsızlığına odaklanmaktadır. Bu amaçla, bu çalışmada H_3 BF'ler için tekdüzeliği ve bağımsızlığı kantitatif olarak tanımlayan muntazam bir çerçeve sunulmaktadır. Buna ek olarak, $H3$ özet fonksiyonlarının çoğuldan tekile oluşları muntazam bir tanım ile sunulmuştur. Önerilen çerçeve daha sonrasında tekdüze ya da bağımsız olma koşulu olmaksızın, $H3$ özet fonksiyonlarına sahip BF'lerin yanlış pozitif olasılıklarının isabetli şekilde hesaplanmasında kullanılmıştır. Bu yanlış pozitif hesabının doğrulaması için

donanım üzerinde saniyede 2×10^8 üyelik kontrolü gerçekleştirebilen bir test yatağı gerçeklenmiştir. Tekdüzeliğin ve bağımsızlığın farklı seviyerlerde kaybının etkilerini incelemek için yapılandırılmış testler gerçekleştirilmiştir. Yanlış pozitif hesabı farklı BF parametre aralıklarında değerlendirilmiştir.

Anahtar Kelimeler: Bloom Filtre, Evrensel Özetleme Fonksiyonları, Yanlış Pozitif Olasılığı, Özet Fonksiyon Tekdüzeliği ve Bağımsızlığı

To my family

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Dr. Ece Güran Schmidt for being the best advisor a researcher could wish for. Her experience and attitude has not only changed me as an engineer but also as a person. For this, I will be grateful for the rest of my days.

This thesis was supported by the Scientific and Research Council of Turkey (TUBITAK) [Project Code 117E667-117E668].

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xiv

LIST OF FIGURES xv

LIST OF ABBREVIATIONS xvi

CHAPTERS

1 INTRODUCTION 1

2 BACKGROUND AND PREVIOUS WORK 5

2.1 An Overview for Bloom Filters and $H3$ Hash Functions 5

2.1.1 Standard BF Implementation with Shared Memory 5

2.1.2 Partioned Memory Implementation of BFs 7

2.1.3 Hash Functions for BFs 9

2.2 Relevant Previous Work 9

3 PRELIMINARIES AND FORMAL NOTATION 17

3.1 Bloom Filters 17

3.2 Bloom Filter False Positive Probability Computation in the Literature 18

3.3 False Positive Probability Computation for the Partitioned Bloom Filters . . . 19

3.4 H_3 (XOR-based) Hash Functions . . . 21

3.5 Vector Spaces over GF(2) . . . 23

4 PROPOSED ANALYTICAL FRAMEWORK FOR PARTITIONED MEMORY BLOOM FILTERS IMPLEMENTED WITH $\boldsymbol{H_3}$ HASH FUNCTIONS 27

4.1 Definition and Conditions for the Uniformity and Independence of H_3 hash functions . . . 27

4.2 Definitions for the Precise False Positive Probability of $\boldsymbol{H_3}$ Hash Functions . . . 30

4.3 Computation for the Precise False Positive Probability of H_3 Hash Functions . . . 32

4.4 Method Derivation . . . 33

4.5 Example Computation of P_{fp} for $k = 2$ . . . 37

4.6 Pseudocode for the Novel False Positive Computation . . . 38

4.7 Algorithmic Complexity . . . 39

5 EVALUATION . . . 43

5.1 Hardware Simulation Environment . . . 43

5.1.1 Problem Definition . . . 43

5.1.2 Testbench Design . . . 44

5.1.3 Implementation . . . 47

5.1.4 Evaluation of Testbench . . . 48

5.2 Evaluation Results . . . 51

5.2.1 Experiment 1 . . . 52

5.2.2 Experiment 2 . . . 55

5.2.3 Experiment 3 57

5.2.4 Experiment 4 58

6 CONCLUSION AND FUTURE WORK 61

REFERENCES 63

LIST OF TABLES

TABLES

Table 5.1 Percentiles for absolute value of e_p. 51

Table 5.2 Test points and results for Experiment 3. 57

Table 5.3 Ratio of $P_{fp}/P_{fp,legacy}$ for $k = 2..10$, $w = 256$ when $dep(H_1, H_2)$ is 0, 10, 15 and 20. 59

LIST OF FIGURES

FIGURES

Figure 2.1 Monolithic BF. 7

Figure 2.2 Partitioned BF. 8

Figure 5.1 Block design of test bench. 45

Figure 5.2 Photograph of the test setup. 48

Figure 5.3 Histogram of percentage errors. 50

Figure 5.4 False positive rates obtained from legacy expression, our expression and tests with respect to $dep(H_1, H_2)$ 53

Figure 5.5 False positive rate ratios obtained from legacy expression, our expression and tests with respect to $dep(H_1, H_2)$ 54

Figure 5.6 False positive rates obtained from legacy expression, our expression and tests with respect to $dep(H_1, H_2, H_3)$ 56

Figure 5.7 False positive rate ratios obtained from legacy expression, our expression and tests with respect to $dep(H_1, H_2, H_3)$ 57

LIST OF ABBREVIATIONS

ABBREVIATIONS	
BF	Bloom Filter
BRAM	Block RAM
CA	Content Advertisement
CAM	Content Addressable Memory
CBF	Counting Bloom Filter
CC	Collision Class
CML	Candidate Mapping Location
CPU	Central Processing Unit
CR	Content Router
CRC	Cyclic Redundancy Check
EBF	Extended Bloom Filter
EC	Equivalence Class
FPGA	Field Programmable Gate Array
FIB	Forward Information Base
GF	Galois Field
GPU	Graphical Processing Unit
ID	Identification
I/O	Input / Output
IP	Internet Protocol
JTAG	Joint Test Action Group
LFSR	Linear Feedback Shift Register
LOFS	Lightweight Online File Storage

MAC	Medium Access Control
MEC	Mobile Edge Computing
MBF	Monolithic Bloom Filter
MD	Message Digest
NIDS	Network Intrusion Detection System
NGS	Next Generation Sequencing
PBF	Partitioned Bloom Filter
PC	Personal Computer
PMH	Parallel Multiple Hashing
SCTP	Stream Control Transmission Protocol
SDN	Software Defined Networking
SEU	Single Event Upset
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator
USB	Universal Serial Bus
VHDL	Very High-Speed Integrated Circuit Hardware Description Language

CHAPTER 1

INTRODUCTION

Membership testing problem can be stated as answering the question of "Is string x an element of a given set $\mathcal{S}$?". Here, $\mathcal{S}$ is sometimes called the *dictionary*.

Membership testing problem emerges in diverse applications including network packet processing [1, 2], table look-up and matching [3, 4, 5, 6, 7, 8, 9, 10],detecting memory address conflicts and errors [11, 12, 13], blockchain applications [14] and human genome mapping[15]. Among these applications, particularly network packet processing and memory access applications require high throughput and low response time calling for fast, low cost and low complexity implementations in hardware.

Bloom Filters (BF) [16] are widely used data structures which can represent the dictionary and query the inputs for set membership. A BF has k hash functions, $h_1 \ldots h_k$ and a bit vector v. Each hash function takes a string x as input and produces an output that is an address for v. Representing the dictionary in the BF is carried out by computing the hash address $\forall x \in \mathcal{S}$ by each h_i and setting the corresponding bit in v. Membership query is performed for a given x by computing the hash addresses for x by each h_i. If all corresponding bits in v are set, the query result is a match indicating $x \in \mathcal{S}$. The hash functions are desired to be uniform and independent to achieve the most efficient use of the memory.

BFs detect all set members correctly and they do not produce false negatives or misses. However, the many-to-one nature of the hashing results in false positive outcomes where a match result is produced for an $x \notin \mathcal{S}$. A false positive result might lead to a wrong address look-up in a router, labeling a legitimate packet as an intrusion and discarding it, or not allowing a legitimate memory access without any

address conflicts. The common remedies for such wrong actions include roll backs and verification of the positive results by using another costly matching engine which degrade the throughput and response time of the membership queries. To this end, Computing the false positive probability of the BF is important for estimating its correctness and performance. *The false positive probability computation in the literature* [17, 5] *assumes that the hash functions of the BF are uniform and independent.*

BFs are implemented on hardware as standalone components or accelerators for high-throughput, low latency applications and querying very large data sets. In such applications, partitioned BFs (PBFs) are employed where each hash function k is assigned an exclusive block of memory to allow concurrent queries. Furthermore, low-complexity hash functions are selected. To this end, *universal* hash functions of $H3$ family are frequently employed because of their low-cost hardware implementation and their capability of producing uniform outputs. The input x and output z of an $H3$ hash function are w bit and y bit row vectors respectively. The output hash address is computed with $z = xH$. Here H is a $w \times y$ Boolean matrix and the jth bit of the output is $z_j = x_{w-1} \cdot h_{w-1,j} \oplus x_{w-2} \cdot h_{w-2,j} \oplus ... \oplus x_0 \cdot h_{0,j}$ [18]. Here we note that, [19, 1, 13, 12, 11] are hardware BF implementations with $H3$ hash functions.

We ask the three following questions in this thesis. Given a partitioned memory BF with k $H3$ hash functions that are implemented as $w \times y$ Boolean matrices. How do we test that these hash functions are uniform and independent such that the legacy false positive probability expression in [17, 5] gives the correct result? Is it possible to construct uniform and independent $H3$ functions for any given k, w and y? What is the precise false positive probability for this PBF if the hash functions are not uniform and/or independent?

This thesis makes the following novel contributions to answer these questions:

- Formal definitions of uniformity and independence for Universal Hash functions.
- A numeric metric that quantifies the uniformity of an $H3$ hash function and the dependence of a set of j $H3$ functions $H_1 \dots H_j$.
- Two theorems that utilize this metric to determine if given $H3$ hash functions

are uniform and/or independent. A result of these theorems is a condition for uniformity and independence for $H3$ hash functions in terms of w, k, y parameters.

- A mathematical framework that formally defines the many-to-one mapping for $H3$ hash functions.
- An exact computation of the false positive probability of a Partitioned Memory Bloom Filter with k $H3$ hash functions where the hash functions are not necessarily uniform and independent.
- A hardware test bench that implements the PBF with desired hash function parameters and performs $2 \cdot 10^8$ queries per second to validate the proposed exact false positive computation.
- An experimental study to demonstrate the results of the proposed false positive computation in comparison with the legacy false positive expression in the literature by analytical evaluation and hardware simulation.

The remainder of this thesis is organized as follows. In Chapter 2 we present an overview for the Bloom Filters and $H3$ hash functions followed by the previous work in the literature. We observe that all the previous work assume that their hash functions are uniform and independent without actually verifying this assumption or considering the effects of violating it. In Chapter 3 we introduce our notation and present the legacy false positive probability computation for Bloom Filters with uniform and independent hash functions. We further present notation and fundamental properties of $GF(2)$ vector (sub)spaces in this chapter that are necessary to build our framework. Chapter 4 presents the formal contributions of this thesis that are listed above. We also present our proofs for the theorems and all formal statements in this chapter. Chapter 5 presents our hardware test bench and then an experimental study. The first part of this study is structured to show the effects of different hash function dependencies on the precise false positive probability. The second part explores a set of hash function parameters and the resulting false positive probability. Chapter 6 summarizes our findings, states our conclusions and plans out the future work for this research.

CHAPTER 2

BACKGROUND AND PREVIOUS WORK

Given a universal set of w-bit strings $\mathcal{U}$ and a set $\mathcal{S} \subset \mathcal{U}$ with the respective cardinalities of $|\mathcal{U}|$ and $|\mathcal{S}|$. Here $|\mathcal{U}| = 2^w$. $\mathcal{S}$ is sometimes called the *dictionary*. For a given $x \in \mathcal{U}$, we call x *positive* if $x \in \mathcal{S}$ and *negative* otherwise. The probability of a string being positive is $P_{\mathcal{S}}$.

A *membership test* represents $\mathcal{S}$ and decides if $x \in \mathcal{S}$ by returning a *match* result.

The applications for membership testing include string matching for network intrusion detection [1, 2], network applications [3, 4, 5, 6, 7, 8, 9], determining address conflicts and error detection in memories [11, 12, 13], file look-ups in large cluster-based storage systems [10], blockchain applications [14] or human genome mapping[15].

2.1 An Overview for Bloom Filters and $H3$ Hash Functions

Bloom Filters (BF) [16, 17, 5] are frequently employed for implementing membership testing applications. BFs are fast and memory-efficient randomized hashing data structures which can represent $\mathcal{S}$ for *approximate* membership testing with possible false positive results.

2.1.1 Standard BF Implementation with Shared Memory

A Bloom Filter (BF) employs k hash functions, $\mathcal{H} = \{h_1, h_2, \ldots, h_k\}$. Each h_i has w bit input, y bit output and maps $\mathcal{U}$ into y bits output space, addressing a memory with a size of $M = 2^y$. We call this implementation with shared memory *Monolithic Bloom Filter* (MBF).

To this end, each string $x \in \mathcal{S}$ is stored in the BF by setting the bit at address in the memory $z = h_i(x)$ computed by each hash function $h_i, i = 1, \ldots, k$.

When an input string x is queried, the BF returns a match result if and only if $h_i(x)$ points to a memory location that is previously set $\forall h_i \in \mathcal{H}$. To this end, there are no misses or false negative results making BFs viable pre-processing engines to eliminate the strings $x \notin \mathcal{S}$.

However, the many to one nature of the hash functions result in *false positives* as it is possible that an input string $x \notin \mathcal{S}$ can be mapped to bits that are set by strings in $\mathcal{S}$. We demonstrate how a false positive outcome can be produced with an example. In this example, $\mathcal{S} = \{S_1, S_2\}$ which is stored in a BF with $k = 2$, $y = 3$ and $M = 2^3$ bits. The queried string $x_1 \notin \mathcal{S}$ falsely produces a match outcome, because the vector positions pointed by the calculated hash values for x_1 are set by S_1 and S_2. Furthermore, $x_2 \notin \mathcal{S}$ is a false positive, as its calculated hash values coincide with the ones of S_2. It is important to state the false positive probability of a given BF because such outcomes can result in performance penalties that manifest as decreased throughput, increased look-up times or wrong actions taken. [12, 17, 5] present the false positive probability Pfp of a BF with k *uniform* and *independent* hash functions that stores $|\mathcal{S}|$ strings in a memory of M bits as:

$$Pfp = \left(1 - e^{-\frac{|\mathcal{S}| \cdot k}{M}}\right)^k. \tag{2.1.1}$$

We present the derivation of Eq. (2.1.1) in Section 3.2. The optimal number of hash functions, k_{opt}, that minimizes Pfp for a given $\mathcal{S}$ and M is computed as in (2.1.2). Then, (2.1.3) follows from (2.1.1) by replacing k with k_{opt}.

$$k_{opt} = \frac{M}{|\mathcal{S}|} \cdot \ln 2, \tag{2.1.2}$$

$$k_{opt} = -\log_2(Pfp). \tag{2.1.3}$$

Here we note that the BF's work with fixed length strings of length w. A number of BFs with different string lengths can be employed to support variable string sizes as in [6]. A number of Bloom filter variants have been proposed that address some of the limitations of the original structure, including counting, deletion, multisets, and space-efficiency [17].

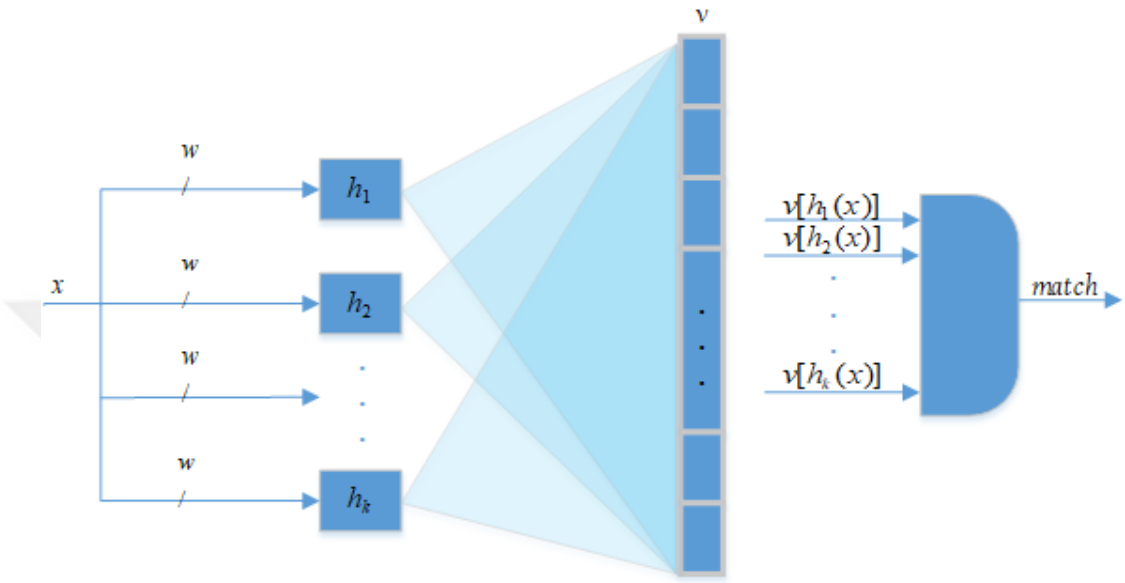


Figure 2.1: Monolithic BF.

2.1.2 Partioned Memory Implementation of BFs

We call the legacy implementation of the BF which features k hash functions that address a single memory block of M bits, *Monolithic Bloom Filter* (MBF). Such implementation requires k read and write ports for the memory to store or query a string in parallel. However, multiple ports significantly increase the memory area cost. It is possible to implement the BF with a single ported memory with serial reads and writes. Such implementation requires multiple cycles and a more complicated control logic. Furthermore large number of hash outputs increase the size of the multiplexers, decoders, and the amount of wiring [12].

An alternative implementation is the *Partitioned Bloom Filter* (PBF) where each hash function h_i, $i = 1, \ldots, k$ has w bit input, y bit output and maps to an exclusive block

of memory of $m = 2^y$ bits. Furthermore, in PBF architecture the hash functions are also less expensive to implement, because they now generate hash values that are smaller by a factor of k. One can show that, if $m = M/k$ the false positive probability of the PBF is asymptotically close to that of an MBF with M bits of memory.

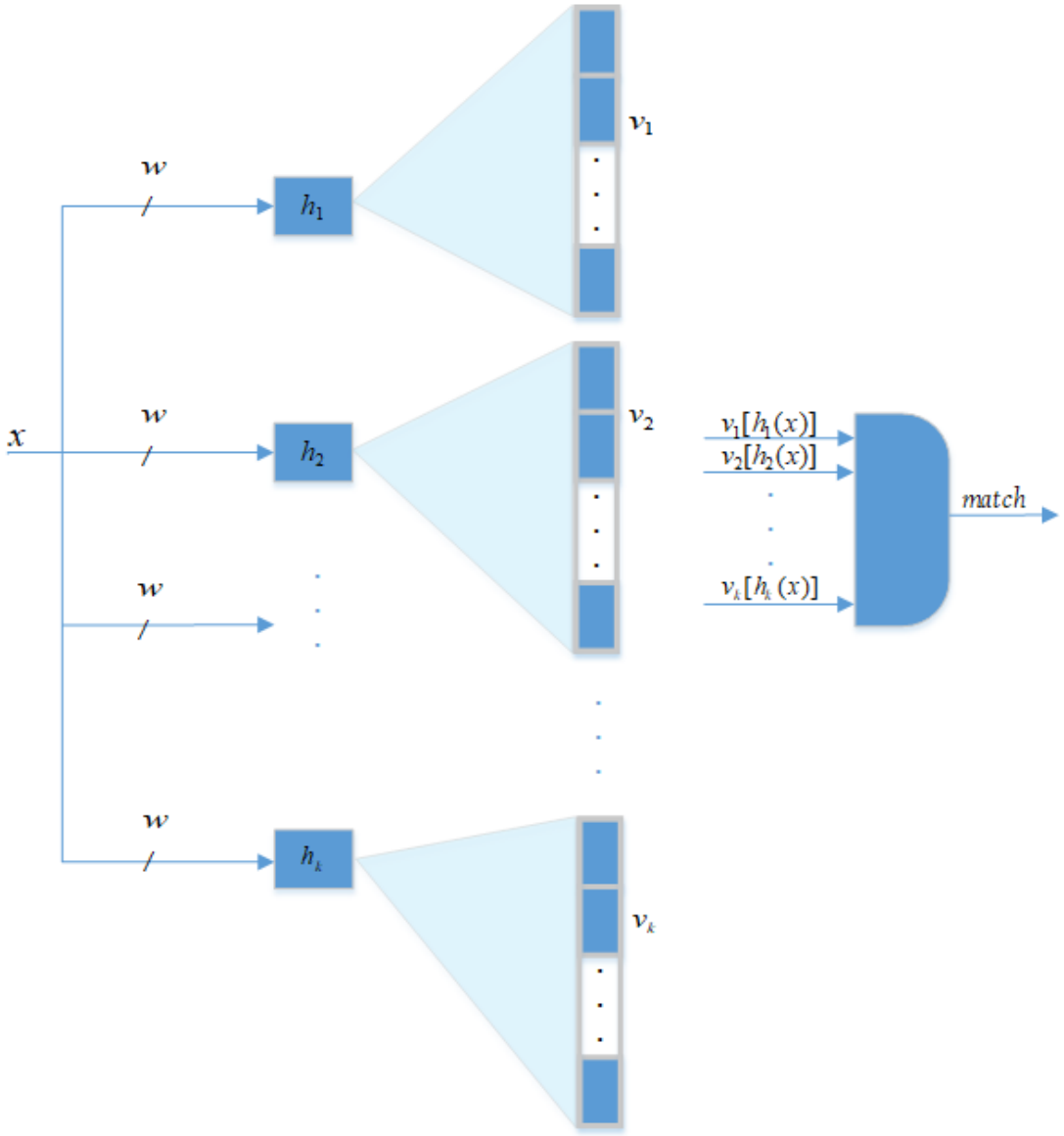


Figure 2.2: Partitioned BF.

2.1.3 Hash Functions for BFs

Previous work on BFs frequently employ $H3$, MD5 [20], CRC32 hash functions [5, 17]. The false positive probability computation of Bloom Filters that we present Eq. (2.1.1) assumes uniform hash functions. The assumption of uniformity is the norm for the analytical model and evaluation of hash-function based schemes [18, 12]. In this work we focus on hardware implementation of hash functions for realizing Bloom Filters. To this end, further desired properties for the hash functions are the capability of realizing different hash functions or dynamically changing the hash function using the same hardware and low implementation complexity to enable high throughput and small latency of a few clock cycles [18, 12, 21].

$H3$ *universal* family of hash functions produce each bit of the y-bit hash output by XOR'ing each bit of the w-bit input string with a preset value of 1 or 0. They distribute the members of $\mathcal{U}$ uniformly to the output address space as they are linear transformations. To this end, $H3$ hash functions can achieve many uncorrelated and uniformly distributed hash values. [18, 12] show that BFs with $H3$ hash functions achieve false positive rates that are close to the analytical expression while other simple hardware hash functions such as bit selection deviate from the analytical model under practical settings. Furthermore, hardwired $H3$ hash functions are relatively inexpensive to implement, requiring a tree of 2-input XOR gates per bit of each hash function. A desired property of the hash function is *surjection* which means that every y bit output is generated by some input $x \in \mathcal{U}$ [21].

2.2 Relevant Previous Work

Next, we present the highly-cited seminal and recent relevant work on different set-membership applications that utilize Bloom Filters and $H3$ hash functions.

[19, 2, 1] are hardware string matching applications for deep packet inspection and intrusion detection systems. In these applications, the detection of certain strings in a stream of network packets indicate an intrusion. The no-match results of the Bloom Filter filter out the strings that are not signatures. The match results are further verified against possible false positives. [19, 1] employ $H3$ hash functions to support the fast hardware implementation. [19, 2] suggest extensions of the fixed-size input Bloom

Filter to variable sized-strings. To this end, [19] implements a dedicated BF for each possible string length. [2] divides longer strings into multiple fixed length shorter segments and uses an Aho-Corasick automaton where the Bloom Filter stores the state transitions. [19, 2] determine k according to the optimal value in Eq. (2.1.2). Their implementation is with a multiport memory. They select $k = 35$ for $|\mathcal{S}| = 10000$, a target $Pfp = 10^{-11}$ and $\frac{M}{|\mathcal{S}|} = 50$. In [2], implementation is with $k = 8$. In [1], architecture features two Bloom Filters. To this end, the first BF works in the legacy manner to filter-out the negatives. The second Bloom Filter caches a small subset of $\mathcal{S}$ which are the recent positive matches. The result of the second Bloom Filter is not verified thanks to its very low false positive probability of 10^{-14}. The design aims for exploiting the temporal locality of input strings where most of the match results are found in the second Bloom Filter. Verification queries are carried out only for the match results of the first Bloom Filter. To this end, the number of verification queries are decreased. They store $|\mathcal{S}| = 9000$ strings of fixed length $w = 208$. Both BF's are implemented with $k = 11$ hash functions where each hash function has $y = 16$ bit output. The implementation environment is SystemC simulation and FPGA synthesis to demonstrate the hardware resource consumption and achievable operating frequency. [2, 1] demonstrate their implementation with the signature sets of Snort [22] for $\mathcal{S}$, which is a popular signature-based NIDS software.

[3, 4] store URLs in BFs. [3] employs Bloom Filters to represent the proxy cache summaries. A proxy server that has a cache miss queries the summaries to find other proxies with possible cache hits. The input strings are URLs and the hash functions are MD5 cryptographic hash functions with 128 bit outputs which are further manipulated by modulus operations. The target false positive ratio is 0.01. The performance tests are done with software simulation. A final implementation is done with modified Squid 1.1.4 software where each BF stores 1 Million URLs with $\frac{M}{|\mathcal{S}|} = 16$, 2 MB memory, $k = 10$, $Pfp = 0.00047$. The average URL length is $w = 400$ bits. [4] proposes URL classification application for Web traffic measurement. They propose a Bloom-filter-based architecture that supports multiple-set membership testing. Their classification problem is stated as finding a group id for each input string $x \in \mathcal{S}$. To this end, a false positive is returning a group ID for some $x \notin \mathcal{S}$ and a misclassification is returning a wrong group for some $x \in \mathcal{S}$. Maximum string length is 256

Bytes. Their proposed data structure needs more hash functions than a typical Bloom Filter. To this end, they suggest using XOR-based hardware hash functions. [4] uses multiple groups of hash functions to differentiate group IDs in one BF. [23] proposes using one group of hash functions to decrease memory access cost. Their target applications are network packet processing for routing and attack detection. The modified BF architecture is implemented with $k = 4, k = 5$ with $Pfp = 0.000049$. The implementation is in software.

[8] aims to represent dynamic sets for network applications including network attack detection, MAC address lookup, IP longest prefix match and multicast routing. To this end, they propose an Extended BF (EBF) that supports both item deletion and expansion. EBF features an elastic bucket array in addition to the BF. The BF is stored in the fast memory and the cooperative bucket array is stored in the slow memory. Every insertion is both in the BF and the bucket array. When the false positive of the BF is above a threshold, the allocated fast memory is increased and rearranged using the information in the bucket array. A reverse procedure is performed for compaction of the BF. The implementation is entirely in software with C++. They use Murmur hash functions with $k = 4, y = 32$. The allocated BF memory is $M = 512MB$. Their max false positive probability that triggers expansion is 0.001 while the minimum false positive probability that triggers compaction is 0.0001. Their strings are IP data traces with 5-tuple information yielding $w = 104$ bits[24]. This 5-tuple consists of; 32-bit IPv4 / 128-bit IPv6 source address, 32-bit IPv4 / 128-bit IPv6 destination address, 16-bit UDP, TCP or SCTP source port number, 16-bit UDP, TCP or SCTP destination port number, 8-bit IPv4/IPv6 protocol number / next header information.

[9, 7] employ Bloom Filters for IP table look-up and evaluate the performance of their architectures using real routing tables [25]. [9] proposes a data structure for binary trie-based IP address lookup where every node is programmed in an on-chip Bloom filter and stored in an offchip hash table. During the look-up, a level-scan is performed by querying the BF until a negative result is produced which indicates that the level of the last positive match is a candidate of the best matching level. Then, the off-chip hash table is probed using the prefix of the candidate level, to verify the result. If there is a false positive, back-tracking is performed by searching at a shorter level. The hash function is a CRC generator. They use an optimal number of hash functions

as indicated in Eq. (2.1.2) with $w = 48bits$ which include 32 bit-prefix, 6-bit length, 8-bit output port, 1 bit valid and 1 bit type fields. They implement 5 routing tables with the following parameters: $(|\mathcal{S}| = 82156, y = 21, k = 18)$, $(|\mathcal{S}| = 191757, y = 22, k = 15)$, $(|\mathcal{S}| = 299899, y = 23, k = 19)$, $(|\mathcal{S}| = 411122, y = 24, k = 28)$, $(|\mathcal{S}| = 576370, y = 24, k = 20)$. The implementation is entirely in software.

[7] proposes Parallel Multiple Hashing (PMH) architecture for trie-based IP address lookup. Each group of prefixes with a distinct length is stored in a separate multihash table. A prefix is stored into the overflow TCAM when the hash table is already full. A BF filters out the length of the input that does not have a matching prefix and eliminates the query to the corresponding hash table. The hash function is with a CRC generator that is composed of shift-right registers with XOR logic. They suggest using optimal k in Eq. (2.1.2) with a target $Pfp = 0.3$. The length of the strings is $w = 38$ bits for 32 bit prefix and 6 bit length information. They implement 5 routing tables with the following parameters:

$(|\mathcal{S}| = 14553, (y = 14, y = 15, k = 2), (y = 16, k = 3), (y = 17, k = 6), (y = 18, k = 11))$, $(|\mathcal{S}| = 39464, (y = 16, k = 2), (y = 17, k = 2), (y = 18, k = 3), (y = 19, k = 6), (y = 20, k = 11))$, $(|\mathcal{S}| = 112310, (y = 17, k = 2), (y = 18, k = 2), (y = 19, k = 3), (y = 20, k = 6), (y = 21, k = 11))$, $(|\mathcal{S}| = 170601, |\mathcal{S}| = 227223, (y = 18, k = 2), (y = 19, k = 2), (y = 20, k = 3), (y = 21, k = 6), (y = 22, k = 11))$.

[26] proposes a multi-field packet classification algorithm for SDN, called hierarchical hash tree (H-HashTree) that is implemented on GPU. An extended Bloom filter is also proposed to accelerate search process by skipping groups in the hash tree. They employ XOR-folded hash functions. There are dedicated BFs for different rule types and the BFs are implemented with $k = 2$ hash functions. The stored string sets are rule sets with $|\mathcal{S}| = 1000, 10000, 100000$ rules where each rule is of size $w = 424$ bits.

[11, 13] use Bloom Filters against memory address conflicts and errors. [11] focuses on Transactional memory systems (TMs) which execute multiple concurrent transactions. When two concurrent transactions perform an access to the same memory address and at least one of the accesses is a write, a conflict occurs. [11] propose

inserting the read/written addresses in a transaction in a BF to track these conflicts. They propose a locality sensitive BF where similar input strings that represent nearby memory locations share some bits of the Bloom filter. To this end, the false positives for transactions that exhibit spatial locality in their read or write sets are reduced without affecting the false positives for transactions that do not exhibit locality at all. The BFs are implemented with partitioned memory. They use $k = 4$ hash functions with $w = 32$ $y = 8, 9, 10, 11, 12, 13$. The implementation environment is GEMS simulator [27] with STAMP[28] benchmark suite. The BF hash functions are $H3$ where each hash function is a surjection without stating any implications about uniformity. They state that the union of matrices by pairs is should be of full rank to achieve different indexes from all hash functions for a given address. We emphasize that statements about surjection or the rank of the matrices are not formal but verbal without any analytical justification.

[13] uses a counting Bloom Filter to mitigate SEU (Single Event Upset) effects in CAM. To this end, the address is fed to a counting Bloom filter and in parallel is given to a CAM. If they both give a miss, the BF eliminates the possibility of SEU for CAM. If CAM output is miss and BF output is a hit, then CAM is checked for SEU. The counting BF features 2 bit counters, and XOR-based hash functions. The string sizes are $w = 32, 64, 128$ bits with $|\mathcal{S}| = [32, 32K]$. The implementation maintains a fixed ($\frac{M}{|\mathcal{S}|} = 8$. They use $k = 3$ hash functions with introduced dependency as in [29]. The target $Pfp = 0.001$. The implementation environment is sim-out order processor simulator [30] with SPECINT benchmarks [31].

[12] uses Cuckoo filter together with Bloom Filters for detecting address conflicts for transactional memory management. Cuckoo filter [32] is another hash-based approximate set-membership testing data structure. They propose to store the read/write set of memory addresses of each transaction in a cuckoo hash table, and to convert this table to Bloom filters when full. They evaluate different hash functions including simple bit selection and $H3$ hash functions. The Bloom filter is implemented with partitioned memory. The implementation parameters are $w = 25$,$k = 4$ and $y = 10$ (per BF). They test upto $|\mathcal{S}| = 1000$. The implementation is with GEMS multiprocessor simulator [27] with a memory modeler [33].

[14] proposes blockchain for network topology protection of Mobile Edge Computing (MEC) systems. Here BFs store information for multidomain collaborative routing consensus without exposing topology privacy. There are multiple domains in MEC systems where each domain usually has a software-defined network (SDN) controller for centralized software-defined control of the underlying network and routing establishment. Each controller generates and maintains a credible access identity to perform the distributed consensus in the blockchain network. To this end, the routing requests are forwarded among controllers together with a BF that stores the routing results for verification of the subsequent controller. This subprocedure will be repeated until the request reaches the destination domain. Finally, the initiating controller receives all the verified paths from multidomain controllers and combines those paths into a trusted multidomain routing. The entire implementation is in software.

[10] proposes a two stage BF architecture for decentralized metadata management in a group of metadata servers. The first stage BF array stores a relatively small amount of frequently accessed metadata. If there is no unique hit in the first stage, the second stage BF array that stores all metadata is queried *sequentially.* The design is evaluated by simulations and a Linux implementation.

[34] proposes a BF-based solution for file deduplication by eliminating the redundant data chunks at network edge file storages. Their Lightweight Online File Storage strategy, LOFS, adopts a three-layer hash mapping scheme where the BF serves as a preprocessing stage. An online-arriving file is partitioned into several variable-sized chunks and then stored into a BF. The suggested hash functions are MD5 [20], SHA-1 [35]. The implementation is in software.

[36] proposes BFs to construct Forward Information Base (FIB) tables for forwarding the Interest packets of the content seekers to the right store in Information Centric Networks (ICN). They employ counting BFs (CBF) with d-left hash function [37]. In this architecture, a Server advertises its content information (Content Advertisement-CA) and Content Routers (CRs) in the network create FIB tables from the information in the advertisement. Based on received CA's the CR prepares the FIB table. The origin server creates a d-shift CBF that contains the names of its contents, and propagates this BF across the network. The information stored in the BF conveys the probability

of availability of the searched content in a path. The implementation is in software with ndnSim-2.0 simulator [38].

[15] focuses on the genome mapping in Next Generation Sequencing (NGS) applications. In this context, short-read mapping is the mapping of DNA subsequences to a known reference genome. They propose a high-throughput hardware accelerator with a Bloom filter-based candidate mapping location (CML) generator for aligning the fixed-length sequence segments (seeds) to a reference genome. A hierarchical BF structure queries a number of seeds in parallel to map a short-read onto a reference segment. The target false positive rate is less than 0.1. They use $H3$ hash functions with partioned memory implementation. The accelerator is realized on a Stratix V GX FPGA with 16GB external SDRAM which operates at 200MHz.

Our literature survey presents a diverse field of applications for Bloom Filters in set-membership problems including very recent academic studies. We observe that $H3$ hash functions are particularly selected for hardware Bloom Filter implementations. All previous work assume that the hash functions are independent and uniform.

To the best of our knowledge, there is no work in the literature on Bloom Filters that verify if their hash functions are indeed uniform and independent or explore the effects on the false positive probability if these requirements are not satisfied.

CHAPTER 3

PRELIMINARIES AND FORMAL NOTATION

In this chapter we introduce our notations and present the required background information for this work that exists in the literature.

3.1 Bloom Filters

Given a universe set of w-bit strings $\mathcal{U}$ and a set $\mathcal{S} \subset \mathcal{U}$ with the respective cardinalities of $|\mathcal{U}| = 2^w$ and $|\mathcal{S}|$. The *membership test* checks if $x \in \mathcal{S}$ is true (positive) or false (negative) for a given $x \in \mathcal{U}$. $\mathcal{S}$ is sometimes called the *dictionary*. We define the probability of a string being positive, $P_{\mathcal{S}}$.

A Bloom Filter (BF) employs k hash functions, $\mathcal{H} = \{h_1, h_2, \ldots, h_k\}$. Each h_i has w bit input, y bit output and maps $\mathcal{U}$ into y bits output space, addressing a memory with a size of $M = 2^y$. To this end, each string $x \in \mathcal{S}$ is stored in the BF by setting the bit at address in the memory $z = h_i(x)$ computed by each hash function $h_i, i = 1, \ldots, k$.

When an input string x is queried, the BF returns a match result if and only if $h_i(x)$ points to a memory location that is previously set, $\forall h_i \in \mathcal{H}$. The many to one nature of the hash functions result in *false positives* as it is possible that an input string $x \notin \mathcal{S}$ can be mapped to bits that are set by strings in $\mathcal{S}$.

We introduce the following conditional probabilities for BFs regarding the *correct-*

ness of the match results:

$$\begin{aligned}
&\text{True positive prob.:} && Ptp = P(\text{match}|\text{positive}) = 1,\\
&\text{False positive prob.:} && Pfp = P(\text{match}|\text{negative}),\\
&\text{True negative prob.:} && Ptn = P(\text{no-match}|\text{negative}),\\
&\text{False negative prob.:} && Pfn = P(\text{no-match}|\text{positive}) = 0.
\end{aligned} \tag{3.1.1}$$

Let P_{match} indicate the probability of a match result of a BF, realized by $h_i(x) = 1$ $\forall h_i \in \mathcal{H}$. We can write the following using Eq.(3.1.1).

$$P_{match} = P(\text{match}|\text{positive}) \cdot P_{\mathcal{S}} + P(\text{match}|\text{negative}) \cdot (1 - P_{\mathcal{S}}) \tag{3.1.2}$$

$$= P_{\mathcal{S}} + Pfp \cdot (1 - P_{\mathcal{S}}). \tag{3.1.3}$$

3.2 Bloom Filter False Positive Probability Computation in the Literature

We first present the analytical computation of false positive probability for the the Monolithic Bloom Filter that we introduce in Section 2.1.1. Here, all k hash functions have w bit inputs, y bit outputs and map to a shared block of memory of $M = 2^y$ bits. We emphasize that the following requirements should be satisfied for these computations:

Assumption 1. Each $h_i \in \mathcal{H}$ is uniform over $M = 2^y$.

Assumption 2. All $h_i \in \mathcal{H}$ are independent.

Assumption 3. $\mathcal{S}$ and the queried strings x are uniformly distributed over $\mathcal{U}$.

The false positive probability Pfp_M of an MBF that stores $|\mathcal{S}|$ strings is [12, 17, 5]:

$$Pfp_M = \left(1 - (1 - \frac{1}{M})^{|\mathcal{S}|\cdot k}\right)^k, \tag{3.2.1}$$

$$\cong \left(1 - e^{-\frac{|\mathcal{S}|\cdot k}{M}}\right)^k \text{ when } \frac{1}{M} \ll 1. \tag{3.2.2}$$

Proof. Let $\rho_M = (1 - \frac{1}{M})^{|\mathcal{S}|\cdot k}$ denote the fraction of bits that are 0 after inserting $|\mathcal{S}|$ strings. Then the probability of a match for $x \in \mathcal{U}$ is $p_{match,M} = (1 - \rho_M)^k$. We denote the false positive probability, $Pfp_M = P(\text{match,M}|\text{negative})$. We can write the following:

$$Pfp_M = p_{match,M} \cdot (1 - P_{\mathcal{S}}) \text{ by [12]},$$
$$p_{match,M} = P_{\mathcal{S}} + Pfp_M \cdot (1 - P_{\mathcal{S}}) \text{ by [1]}.$$

Here one should note that under very small $P_{\mathcal{S}}$ ($|\mathcal{S}| \ll |\mathcal{U}|$), $Pfp_M \cong p_{match,M}$ resulting in $Pfp_M = (1 - \rho_M)^k$.

Taylor Series expansion of the exponential function $e^x = \sum_{n=0}^{\infty} \frac{1}{n!}x^n$, under $\left|\frac{1}{M}\right| \ll 1$ yields $e^{-1/M} \cong 1 - \frac{1}{M}$. Then; $\rho_M \cong e^{-\frac{|\mathcal{S}|\cdot k}{M}}$ and $Pfp_M \cong (1 - e^{-\frac{|\mathcal{S}|\cdot k}{M}})^k$ when $\frac{1}{M} \ll 1$ [12].

□

3.3 False Positive Probability Computation for the Partitioned Bloom Filters

The focus of this thesis is the partitioned Bloom Filters (PBF) that we introduce in Section 2.1.2. To this end, each $h_i \in \mathcal{H}$ maps $\mathcal{U} \mapsto \{0, 1, \ldots, 2^y - 1\}$ using an exclusive block of memory with a size of $m = 2^y$. The amount of total memory is $M = k \cdot m$.

The false positive probability $P_{fp,legacy}$ in Eq (3.3.1) of a PBF that stores $|\mathcal{S}|$ strings

is computed under the 3 Assumptions listed in Section 3.2, with a modification of Assumption 1. To this end, each $h_i \in \mathcal{H}$ is uniform over $m = 2^y$.

$$P_{fp,legacy} = \left(1 - (1 - \frac{1}{m})^{|\mathcal{S}|}\right)^k. \tag{3.3.1}$$

Proof. Let ρ_m denote the fraction of bits that are 0 after inserting $|\mathcal{S}|$ strings. We follow an analogous approach to the derivation of Pfp_M and write $\rho_m = (1 - \frac{1}{m})^{|\mathcal{S}|}$. Then, $P_{fp,legacy} = p_{match,m} = (1 - \rho_m)^k$.

We apply the same Taylor Series expansion of the exponential function e^x under $\frac{k}{m} \ll 1$, $1 - \frac{1}{m/k} \cong e^{-\frac{k}{m}}$. This approximation yields, $\rho_m \cong e^{-\frac{|\mathcal{S}|}{m}}$ and $P_{fp,legacy} \cong (1 - e^{-\frac{|\mathcal{S}|}{m}})^k$ under $\frac{k}{m} \ll 1$ [12].

□

Remark 3.1. A PBF with k hash functions and $m = M/k$ has asymptotically the same false positive probability as an MBF. However, since $k \geq 1$, $\left(1 - \frac{1}{M/k}\right)^{|\mathcal{S}|} \leq \left(1 - \frac{1}{M}\right)^{k \cdot |\mathcal{S}|}$ the false positive probability of the PBF with the same amount of memory as an MBF is always larger. This can also be intuitively be explained with the larger fill factors of the PBF because of the M/k range restriction[17, 5].

Remark 3.2. In this work, we employ PBFs for fast BF implementations. To this end, $P_{fp,legacy}$ is represented in exact form in Eq. (3.3.1). We do not consider the corresponding MBF with equal amount of memory and treat the exclusive memory per hash function m as a free design parameter. From this point on, we will refer to the expression given in Eq. (3.3.1) as the legacy false positive expression, $P_{fp,legacy}$.

Remark 3.3. In this work, we consider the shortcomings of this legacy implementation. More specifically, we elaborate on what happens when uniformity and independence requirements are not met. We expect that deviations from these requirements will result in higher false positive rates than the estimation obtained by the legacy expression. Consider following exaggerated examples:

1. Violating uniformity: Consider a BF where each hash function maps all inputs

to the same output. It's clear that this BF will perform a lot worse than what is expected by the legacy implementation.

2. Violating independence: Consider another BF with $k = 2$ hash functions where two hash functions are identical. Clearly, this bloom filter will behave as if $k = 1$ and considering (3.3.1), this deviation in independence will degrade performance.

It is evident deviations from these assumptions have adverse effects. When deviations are not as extreme, the consequences will be less significant yet still exist.

3.4 H_3 (XOR-based) Hash Functions

In their work, [39] define $universal_2$ hash function family H as follows:

Definition 3.1. Universal Hash Functions
Hash function family, H is a class of functions from $\mathcal{A}$ to $\mathcal{B}$ with $|\mathcal{A}| > |\mathcal{B}|$. H is $universal_2$, if no pair of distinct keys collide under more than $\frac{1}{|\mathcal{B}|}$ fraction of the functions in H. Consequently;

$$\forall p, q \in \mathcal{A}, p \neq q : \Pr_{h \in H}[h(p) = h(q)] \leq \frac{1}{|\mathcal{B}|} \tag{3.4.1}$$

Note that, this definition only considers pairwise collision probability and does not guarantee the uniformity of an individual hash function.

H_3 hash functions are proven to be $universal_2$ in [39] and defined as follows:

Definition 3.2. H_3 Universal Hash Function Class

- $\mathcal{Q}$: the set of all possible $w \times y$ with $w > y$, Boolean matrices constitute H_3 class.
- $H \in \mathcal{Q}$ is a $w \times y$ Boolean matrix represented with :

$$H = \begin{bmatrix} h_{w-1,y-1} & h_{w-1,y-2} & \dots & h_{w-1,0} \\ h_{w-2,y-1} & h_{w-2,y-2} & \dots & h_{w-2,0} \\ \vdots & \ddots & \ddots & \vdots \\ h_{0,y-1} & h_{0,y-2} & \dots & h_{0,0} \end{bmatrix} \in GF(2)^{w \times y}$$

- An input string $x \in \mathcal{U}$ is represented with a w-bit row vector, $x = \begin{bmatrix} x_{w-1} & x_{w-2} & \dots & x_0 \end{bmatrix} \in GF(2)^{1 \times w}$
- Hash output z is a y-bit binary output that is represented with y bit row vector, $z = \begin{bmatrix} z_{y-1} & z_{y-2} & \dots & z_0 \end{bmatrix} \in GF(2)^{1 \times y}$

The hash address z is computed with the following vector matrix multiplication in $GF(2)$:

$$xH = z. \tag{3.4.2}$$

Here $z_j = x_{w-1} \cdot h_{w-1,j} \oplus x_{w-2} \cdot h_{w-2,j} \oplus ... \oplus x_0 \cdot h_{0,j}$ where $\cdot$ denotes bit-wise AND and $\oplus$ denotes bit-wise exclusive OR operations according to vector - matrix multiplication in $GF(2)$.

In the example below, H computes a 2-bit address $z = \begin{bmatrix} z_1 & z_0 \end{bmatrix}$ for a 3-bit input string x.

$$\begin{bmatrix} x_2 & x_1 & x_0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} x_2 \oplus x_1 & x_2 \oplus x_0 \end{bmatrix}$$

The hashing functions from the class H_3 are essentially linear transformations from 2^w to 2^y and they distribute each $x \in \mathcal{U}$ into the address range with the decimal equivalent $\{0, 1, \dots, 2^y - 1\}$.

Remark 3.4. For the rest of this work, all hash functions are H_3 class expressed as a $w \times y$ Boolean matrix.

3.5 Vector Spaces over GF(2)

We next provide the relevant definitions for GF(2) and then subsequently, the derived properties of the H_3 hash functions expressed as Boolean matrices.

A field is defined as a set with two binary operations addition and multiplication. For a field, both of the binary operations are commutative and associative. Moreover, identity elements and inverses are defined for these binary operations. A field with a finite number of elements is usually called a finite field, or Galois field GF. GF(2) is the finite field with two elements If $\mathcal{F} = \{0, 1\}$ where addition and multiplication operations are defined with the Boolean AND ($\cdot$) and XOR ($\oplus$) the vector space over GF(2) is constructed according to Boolean Algebra. To this end, many uses of $GF(2)$ vector algebra exist in computer engineering particularly in coding theory. We make the following definitions and state the relevant properties of $GF(2)$ vector algebra [21].

Definition 3.3. $GF(2)^w$ **vector space**
A $GF(2)^w$ vector (sub)space with dimension n is a set $\mathcal{V}$ of w-bit vectors with $|\mathcal{V}| = 2^n$.

Properties:

P3.3.1: Every vector space has the zero vector.

P3.3.2: The space is closed under $\oplus$.

P3.3.3: A vector space has a set of basis vectors $\{v_1, v_2, \ldots, v_n\}$ which are linearly independent. $\alpha_1 \cdot v_1 \oplus \alpha_2 \cdot v_2 \oplus \ldots \oplus \alpha_n \cdot v_n \neq 0$, $\alpha_i \in \{0, 1\}$, $\exists \alpha_i = 1$.

P3.3.4: $\mathcal{V} = span(v_1, v_2, \ldots, v_n)$. The basis vectors *span* the vector space. Any $x \in \mathcal{V}$ has a unique decomposition as a linear combination of the basis vectors.

Definition 3.4. Null space and Column Space of matrix H
The left null space $N(H)$ of a matrix H is a vector space which is defined as the set of vectors $\{x \ : \ xH = \mathbf{0}\}$.
The column space $col(H)$ of a matrix is H is a vector space defined as $col(H) = span(c_1, c_2, \ldots, c_y)$ where $c_{1..y}$ are the columns of matrix H.

Properties:

P3.4.1: The column space and null space are orthogonal complements i.e. $col(H) \perp$

$N(H)$ for finite vector space $GF(2)^w$. By the definition of orthogonal complement, $N(H)$ contains all vectors that are orthogonal to every vector in $col(H)$.

P3.4.2: $dim(col(H)) \leq y$ as $w > y$. To this end we use $dim(H) \triangleq dim(col(H))$ for the remainder of this thesis.

P3.4.3: $w = dim(N(H)) + dim((H))$ (Rank and Nullity Theorem).

Definition 3.5. H_3 **hash function surjection** [21]

Given a *surjective* H. All possible hash addresses $z \in \{0, 1, \ldots, 2^y - 1\}$ are mapped by some $x \in \mathcal{U} \iff dim(H) = y$.

Definition 3.6. Equivalence Class The equivalence class $EC_i(x)$ for input x and hash function H_i is the set of inputs that are mapped to same output by H_i[21]:

$$EC_i(x) \triangleq \{q \mid xH_i = qH_i, \quad \forall q \in U\} \text{ with} \\ EC_i(x) = EC_i(q) \tag{3.5.1}$$

Properties:

P3.6.1: $xH_i = qH_i \iff x \oplus v = q, v \in N(H_i)$ (3.5.2)

Proof.

$$\begin{aligned} x \oplus v = q, v \in N(H_i) \Rightarrow & (x \oplus v)H_i = qH_i \\ \Rightarrow & xH_i \oplus vH_i = qH_i \\ \Rightarrow & xH_i = qH_i \end{aligned}$$

$$\begin{aligned} xH_i = qH_i, x \oplus v = q \Rightarrow & v \in N(H_i) \\ & (x \oplus v)H_i = qH_i \\ \Rightarrow & xH_i \oplus vH_i = qH_i \\ \Rightarrow & vH_i = 0 \\ \Rightarrow & v \in N(H_i). \end{aligned}$$

□

P3.6.2: $xH_i = qH_i \iff x \oplus q = v, v \in N(H_i)$ (3.5.3)

Proof.

$$\begin{aligned} xH_i = qH_i \Rightarrow & x \oplus v = q, v \in N(H_i) \\ \Rightarrow & x \oplus x \oplus v = x \oplus q \\ \Rightarrow & v = x \oplus q \end{aligned}$$

$$\begin{aligned} x \oplus q = v, v \in N(H_i) \Rightarrow & xH_i = qH_i \\ \Rightarrow & x \oplus q \oplus q = v \oplus q \\ \Rightarrow & x = v \oplus q \\ \Rightarrow & xH_i = qH_i \end{aligned}$$

□

P3.6.3: Equivalence classes are the only reason of collision for H_3 type of hash functions [21].

P3.6.4: Each equivalence class have the same size and shape defined by $N(H)$ (Property 7.1,[21]).

P3.6.5: Two hash functions h_i and h_j are equivalent if and only if $N(H_i) = N(H_j)$. Note that h_i and h_j may have different outputs for the same inputs.

CHAPTER 4

PROPOSED ANALYTICAL FRAMEWORK FOR PARTITIONED MEMORY BLOOM FILTERS IMPLEMENTED WITH H_3 HASH FUNCTIONS

In the scope of this thesis we focus on partitioned Bloom Filters implemented with H_3 hash functions according to Definition 3.2 presented in Section 3.4. This chapter presents the analytical framework that is entirely developed in the scope of this thesis.

4.1 Definition and Conditions for the Uniformity and Independence of H_3 hash functions

Definition 4.1. Hash function uniformity: Given $x \in \mathcal{U}$ and hash function h with y outputs. h is uniform over $\mathcal{U}$ if:

$$\underset{x \in \mathcal{U}}{P}[h(x) = z] = \frac{1}{2^y}, \forall x \in \mathcal{U}, \forall z \in \{0, 1, \ldots, 2^y - 1\}.$$

Definition 4.2. Independence of hash functions: Given $x \in \mathcal{U}$ and hash functions $h_1, h_2, ..., h_j$ are independent if:

$$\underset{x \in \mathcal{U}}{P}[h_1(x) = z_1 \wedge\ h_2(x) = z_2\ \wedge \ldots \wedge h_j(x) = z_j] =$$
$$\underset{x \in \mathcal{U}}{P}[h_1(x) = z_1] \cdot \underset{x \in \mathcal{U}}{P}[h_2(x) = z_2] \cdot \ldots \cdot \underset{x \in \mathcal{U}}{P}[h_j(x) = z_j] = \frac{1}{2^{j \cdot y}}, \tag{4.1.1}$$
$$\forall z_i \in \{0, 1, \ldots, 2^y - 1\}, i \in \{1, \ldots, j\}$$

Remark 4.1. Given a hash function h with a $w \times y$ matrix H, $rank(H) = dim(H) = dim(col(H))$. For h to be surjective, it is required that $dim(col(H)) = y$, i.e, all columns of h are independent [21]. By the rank and nullity theorem in P3.4.3. maximizing the dimension of the column space up to y, number of inputs that are mapped to the same output can be decreased.

Theorem 1. Uniformity of H_3 Hash Functions

Given a hash function h that belongs to H_3 family that is represented as a $w \times y$ Boolean matrix H with $w > y$.

$$h \text{ is uniform over } |\mathcal{U}| = 2^w \iff dim(H) = y.$$

Proof

$\Rightarrow$ Definition 4.1 states that all 2^y hash addresses are mapped by an equal (and non-zero) number of strings in $\mathcal{U}$. To this end, H should be surjective with $dim(H) = y$ by Definition 3.5 [21].

$\Leftarrow$ The cardinality of $EC_i(x)$ is $|EC_i(x)| = 2^{dim(N(H_i))} = 2^{w-dim(H_i)}$ by P3.6.4.

The number of ECs for H_i is $\frac{2^w}{2^{dim(N(H_i))}} = 2^{dim(H)}$ by P3.6.4. and P3.4.3

Now, pick an arbitrary $x \in \mathcal{U}$. It holds that x belongs to a unique EC. Since the strings in each EC are mapped to a unique address (this should be known from somewhere), this means that x is mapped to some address $z \in \{0, 1, \ldots, 2^y\}$. That is, indeed, P$[h(x) = z] = \frac{1}{2^y}$, which shows that h_i is uniform over $\mathcal{U}$. This is also the number of hash addresses that are are mapped by some $x \in \mathcal{U}$.

if $dim(H) = y$ then 2^y addresses are mapped.

Properties:

P4.2.1: $dim(H_1, H_2, ..., H_j) \leq min(w, jy)$ is the dimension of the subspace spanned by columns of $H_1, H_2, ..., H_j$.

P4.2.2: $dim(H_1, H_2, \ldots, H_j, H_{j+1}) \leq dim(H_1, H_2, \ldots, H_j) + dim(H_{j+1})$

$\leq dim(H_1, H_2, \ldots, H_j) + y$. We define the following metric.

Definition 4.3. Dependence Metric: Given hash functions $h_1, h_2, ..., h_j$. $dep(H_1, H_2, \ldots H_j) \triangleq j \cdot y - dim(H_1, H_2, \ldots H_j)$

Theorem 2. Independence of H_3 Hash Functions

a)$dep(H_1, H_2, \ldots, H_j) = 0 \Rightarrow$ Each of the hash functions $h_1, h_2, ..., h_j$ are uniform

b)$dep(H_1, H_2, \ldots, H_j) = 0 \iff$ Hash functions $h_1, h_2, ..., h_j$ are independent.

Proof. **(a):** $dep(H_1, H_2, \ldots, H_j) = 0 \Rightarrow$ Each of the hash functions $h_1, h_2, ..., h_j$ are uniform.

By Definition 4.3, $dep(H_1, H_2, \ldots, H_j) = 0 \iff dim(H_1, H_2, \ldots, H_j) = j \cdot y$

By P4.2.2, $dim(H_1, H_2, \ldots, H_j) = j \cdot y \Rightarrow dim(H_1) = y \wedge dim(H_2) = y \wedge \ldots \wedge dim(H_j) = y$

By Theorem 1, $dim(H_1) = y \wedge dim(H_2) = y \wedge \ldots \wedge dim(H_j) = y \iff$ Each of the hash functions $h_1, h_2, ..., h_j$ are uniform.

(b): $dep(H_1, H_2, \ldots, H_j) = 0 \iff$ Hash functions $h_1, h_2, ..., h_j$ are independent.

$\Rightarrow$: We assume that $dep(H_1, H_2, \ldots, H_j) = 0$. That is, by Definition 4.3, it follows that $dim(H_1, H_2, \ldots, H_j) = j \cdot y$. Specifically, the matrix $[H_1 H_2 \cdots H_j]$ that is obtained by horizontally appending all matrices $H_1, H_2, \ldots, H_j$ has rank $j \cdot y$ and is hence uniform. Using Theorem 1, this implies that

$$\frac{1}{2^{j \cdot y}} = \Pr_{x \in \mathcal{U}}[h_1(x) = z_1 \wedge\ h_2(x) = z_2\ \wedge \ldots \wedge h_j(x) = z_j] =$$
$$\Pr_{x \in \mathcal{U}}[h_1(x) = z_1] \cdot \Pr_{x \in \mathcal{U}}[h_2(x) = z_2] \cdot \ldots \cdot \Pr_{x \in \mathcal{U}}[h_j(x) = z_j],$$
$$\forall z_i \in \{0, 1, \ldots, 2^y - 1\}, i \in \{1, \ldots, j\}$$

Hence, the Hash functions $h_1, h_2, \ldots, h_j$ are independent in line with Definition 4.2.

$\Leftarrow$: We assume that the Hash functions $h_1, h_2, ..., h_j$ are independent. Now let $dep(H_1, H_2, \ldots, H_j) > 0$. That is, $dim(H_1, H_2, \ldots, H_j) < j \cdot y$. There are two possible cases. In the first case, $dim(H_i) < y$ for some $i \in \{1, \ldots, j\}$. However, this implies that $\Pr_{x \in \mathcal{U}}[h_i(x) = z_i] \neq \frac{1}{2^y}$ by Theorem 1, which contradicts the assumption that

$$\Pr_{x \in \mathcal{U}}[h_1(x) = z_1] \cdot \Pr_{x \in \mathcal{U}}[h_2(x) = z_2] \cdot \ldots \cdot \Pr_{x \in \mathcal{U}}[h_j(x) = z_j] = \frac{1}{2^{j \cdot y}},$$
$$\forall z_i \in \{0, 1, \ldots, 2^y - 1\}, i \in \{1, \ldots, j\}$$

In the second case, $dim(H_i) = y$ for all $i \in \{1, \ldots, j\}$ but $dim([H_1 H_2 \cdots H_j]) < j \cdot y$. Again, using Theorem 1, this contradicts the assumption that

$$\Pr_{x \in \mathcal{U}}[h_1(x) = z_1 \wedge\ h_2(x) = z_2\ \wedge \ldots \wedge h_j(x) = z_j] = \frac{1}{2^{j \cdot y}},$$
$$\forall z_i \in \{0, 1, \ldots, 2^y - 1\}, i \in \{1, \ldots, j\}$$

Together, this proves that the assumption

$$dep(H_1, H_2, \ldots, H_j) > 0$$

leads to a contradiction. Hence, it must be the case that $dep(H_1, H_2, \ldots, H_j) = 0$.

□

Corollary 2.1. Given $dep(H_{j+1}) = d$, $dep(H_1, H_2, \ldots H_j) = a$
$\Rightarrow dep(H_1, H_2, \ldots H_j, H_{j+1}) \geq (d + a)$.

Proof. By Definition 4.3, $dim(H_{j+1}) = y - d$ and $dim(H_1, H_2, \ldots H_j) = j \cdot y - a$.
By P4.2.2, $dim(H_1, H_2, \ldots, H_j, H_{j+1}) \leq dim(H_1, H_2, \ldots, H_j) + dim(H_{j+1}) \leq (j \cdot y - a) + y - d \leq (j+1) \cdot y - (d+a)$.
By Definition 4.3, $dep(H_1, H_2, \ldots H_j, H_{j+1}) = (j+1) \cdot y - dim(H_1, H_2, \ldots H_{j+1})$
$dep(H_1, H_2, \ldots H_j, H_{j+1}) \geq (j+1) \cdot y - ((j+1) \cdot y - (d+a)) \geq d + a$

□

4.2 Definitions for the Precise False Positive Probability of H_3 Hash Functions

Definition 4.4. Collision Class The collision class $CC_i(x)$ for input x and hash function H_i is the set of inputs other than x that are mapped to the same output by h_i.

$$CC_i(x) \triangleq EC_i(x) \setminus \{x\} \tag{4.2.1}$$

By its definition, the size of a collision class is:

$$|CC_i(x)| = |EC_i(x)| - 1 = 2^{w - dim(H_i)} - 1 \tag{4.2.2}$$

Definition 4.5. Intersection of ECs in H_3 BF
In Section 3.5, the behaviour of a single hash function is explained through $GF(2)$ vector algebra. Here we use a similar approach to elaborate on the combined behaviour of multiple hash functions.
Properties:
P4.5.1: Strings in the intersection of the ECs of multiple hash functions are mapped to the same output for all of these hash functions:

$$\begin{gathered} \text{Let } x, q \in \mathcal{U} \\ q \in (EC_1(x) \cap EC_2(x) \cap \ldots \cap EC_j(x)) \iff \\ h_1(x) = h_1(q) \ \wedge \ h_2(x) = h_2(q) \ \wedge \ldots \wedge \ h_j(x) = h_j(q) \end{gathered} \tag{4.2.3}$$

Proof. By definition of set intersection,

$$\begin{aligned} & q \in (EC_1(x) \cap EC_2(x) \cap \ldots \cap EC_j(x)) \\ \iff & q \in EC_1(x) \ \wedge \ q \in EC_2(x) \ \wedge \ldots \wedge \ q \in EC_j(x) \end{aligned}$$

By definition of equivalence classes in Eq. (3.5.1),

$$q \in EC_1(x) \ \wedge \ q \in EC_2(x) \ \wedge \ldots \wedge \ q \in EC_j(x)$$
$$\iff h_1(x) = h_1(q) \ \wedge \ h_2(x) = h_2(q) \ \wedge \ldots \wedge \ h_j(x) = h_j(q)$$

Hence,

$$q \in (EC_1(x) \cap EC_2(x) \cap \ldots \cap EC_j(x))$$
$$\iff h_1(x) = h_1(q) \ \wedge \ h_2(x) = h_2(q) \ \wedge \ldots \wedge \ h_j(x) = h_j(q)$$

□

P4.5.2: $|EC_1(x) \cap EC_2(x) \cap \ldots \cap EC_j(x)| = 2^{w-dim(H_1,H_2,...,H_j)}, \quad \forall x \in \mathcal{U}$ (4.2.4)

where $dim(H_1, H_2, ..., H_j)$ is the dimension of the space spanned by columns of $H_1, H_2, ..., H_j$.

Proof. Using properties P3.6.1, P3.6.3 and P4.5.1 one can write the following:

$$\forall q \in EC_1(x) \cap EC_2(x) \cap \ldots \cap EC_j(x)$$
$$x \oplus v = q, \ \ v \in N(H_1) \cap N(H_2) \cap \ldots \cap N(H_j)$$

Hence,

$$|EC_1(x) \cap EC_2(x) \cap \ldots \cap EC_j(x)| = |N(H_1) \cap N(H_2) \cap \ldots \cap N(H_j)| \quad (4.2.5)$$

Note that null space itself is also an equivalence class which is mapped to output **0**. Therefore P4.5.1 applies to null spaces. Adding to that, the definition of null spaces, one can write the following:

$$v \in N(H_1) \cap N(H_2) \cap \ldots \cap N(H_j) \iff vH_1 = vH_2 = \ldots = vH_j = \mathbf{0}$$

$$col(H_1, H_2, \ldots, H_j)^{\perp} = N(H_1) \cap N(H_2) \cap \ldots \cap N(H_j)$$

where $col(H_1, H_2, \ldots, H_j)$ is the subspace spanned by columns of matrices $H_1, H_2, \ldots, H_j$ and $N(H_1) \cap N(H_2) \cap \ldots \cap N(H_j)$ is its orthogonal complement, therefore,

$$|N(H_1) \cap N(H_2) \cap \ldots \cap N(H_j)| = |EC_1(x) \cap EC_2(x) \cap \ldots \cap EC_j(x)|$$
$$= 2^{w-dim(H_1,H_2,...,H_j)}$$

□

P4.5.3: $CC_1(x) \cap CC_2(x) \cap \ldots \cap CC_j(x) =$ (4.2.6)

$$(EC_1(x) \cap EC_2(x) \cap \ldots \cap EC_j(x)) \ \setminus \ \{x\}, \quad \forall x \in \mathcal{U}$$

Proof. First we introduce the following property of set algebra: For any set A, B and C where C^{c} is the complement of set C.

$$\begin{aligned}(A \cap B)\backslash C &= A \cap B \cap C^{\mathsf{c}} \\ &= A \cap B \cap C^{\mathsf{c}} \cap C^{\mathsf{c}} \\ &= (A \cap C^{\mathsf{c}}) \cap (B \cap C^{\mathsf{c}}) \\ &= (A\backslash C) \cap (B\backslash C)\end{aligned}$$

Using the definition of collision classes in Eq. (4.2.1) and aforementioned property for sets,

$$\begin{aligned}&CC_1(x) \cap CC_2(x) \cap \ldots \cap CC_j(x) \\ &= (EC_1(x)\backslash\{x\}) \cap (EC_2(x)\backslash\{x\}) \cap \ldots \cap (EC_j(x)\backslash\{x\}) \\ &= ((EC_1(x) \cap EC_2(x))\backslash\{x\}) \cap \ldots \cap (EC_j(x)\backslash\{x\}) \\ &\vdots \\ &= (EC_1(x) \cap EC_2(x) \cap \ldots \cap EC_j(x)) \ \setminus \ \{x\}\end{aligned}$$

□

P4.5.4: $|CC_1(x) \cap CC_2(x) \cap \ldots \cap CC_j(x)| =$ (4.2.7)

$$|EC_1(x) \cap EC_2(x) \cap \ldots \cap EC_j(x)| - 1, \quad \forall x \in \mathcal{U}$$

Remark 4.2. For $j = 1$, $dep(H_1) = y - dim(H_1)$. To this end, when used with individual hash functions, we use $dep(.)$ as a metric to quantify the uniformity for the remainder of this thesis.

4.3 Computation for the Precise False Positive Probability of H_3 Hash Functions

Here we present a novel expression for false positive probability of H_3 PBFs that does not suffer from the following shortcomings of the legacy expression:

- Assumption of hash uniformity

- Assumption of hash independence

As explained in Section 3.3, the legacy expression focuses on the fraction of bits that are set in the memory vectors after the programming and the probability of mapping a random input x into the fractions that are set to 1. Whereas, in this work we focus on the behaviour of the hash functions and the probability of existence for a member string $s \in \mathcal{S}$ in collision classes for a given random input x.

For the purposes of this work we redefine the following:

- **Match probability** (P_{match})**:** Given random input x, what is the probability that at all of the k hash function outputs are mapped by at least one member string $s \in \mathcal{S}$?

- **False positive probability** (P_{fp})**:** Given random input x that is *not a member*, what is the probability that at all of the k hash function outputs are mapped by at least one member string $s \in \mathcal{S}$?

Note that assuming $|\mathcal{S}| << |\mathcal{U}|$ these probabilities are nearly equal. In this work, we refrain from this assumption and directly focus on false positive probability P_{fp}.

4.4 Method Derivation

An input string x is mapped to the same location only with members of its equivalence class (P3.6.3). In addition, it is given in the false positive probability definition that $x \notin \mathcal{S}$. Considering these two properties, the following is derived for a given $x \notin \mathcal{S}$, and $s \in \mathcal{S}$:

$$h_i(x) = h_i(s) \iff CC_i(x) \cap S \neq \emptyset, \tag{4.4.1}$$

$$h_i(x) \neq h_i(s) \iff CC_i(x) \cap S = \emptyset. \tag{4.4.2}$$

PBF yields a match result for an input x if all of the memory locations addressed by hash functions are set by a member string $s \in \mathcal{S}$. Therefore, for this non-member x, a false positive result does not occur when at least one of the hash functions have $h_i(x) \neq h_i(s), \forall s \in \mathcal{S}$.

Considering the conditional expression of false positive probability, the precedent requires us to use $CC_i(x) \triangleq EC_i(x) \setminus \{x\}$, because if the shared element of $EC_i(x)$ and $\mathcal{S}$ is x, this would violate the precedent that is *(*x is not a member).

Hence, using Eq. (4.4.2) we can define the following:

$$1 - P_{fp} = \underset{x \in U}{\mathrm{P}} \left[\bigvee_{i=1}^{k} CC_i(x) \cap S = \emptyset \right] = \tag{4.4.3}$$
$$\underset{x \in \mathcal{U}}{\mathrm{P}} [CC_1(x) \cap \mathcal{S} = \emptyset \ \vee \ CC_2(x) \cap \mathcal{S} = \emptyset \ \vee \cdots \vee \ CC_k(x) \cap \mathcal{S} = \emptyset]$$

Logically ORed events in the probability expression in Eq. (4.4.3) are not necessarily independent. To this end, we use inclusion & exclusion [40] principle which is a counting method for the number of members in set unions. Accordingly, we divide the expression into simpler components as follows:

$$\begin{aligned} 1 - P_{fp} = & \sum_{i=1}^{k} \underset{x \in U}{\mathrm{P}} [CC_i(x) \cap S = \emptyset] \\ & - \sum_{1 \leq i < j \leq k} \underset{x \in U}{\mathrm{P}} [CC_i(x) \cap S = \emptyset \ \wedge \ CC_j(x) \cap S = \emptyset] \\ & + \sum_{1 \leq i < j < t \leq k} \underset{x \in U}{\mathrm{P}} [CC_i(x) \cap S = \emptyset \ \wedge \ CC_j(x) \cap S = \emptyset \ \wedge \ CC_t(x) \cap S = \emptyset] \\ & \cdots \\ & + \ (-1)^{k-1} \underset{x \in U}{\mathrm{P}} [CC_1(x) \cap S = \emptyset \ \wedge \ CC_2(x) \cap S = \emptyset \ \wedge \ \ldots \ \wedge \ CC_k(x) \cap S = \emptyset] \end{aligned} \tag{4.4.4}$$

In order to compute probabilities given above individual additive components in Eq. (4.4) can be expressed as follows:

$$\underset{x\in U}{\mathbf{P}}[CC_1(x)\cap S=\emptyset \ \wedge \ CC_2(x)\cap S=\emptyset \ \wedge \ \ldots \ \wedge \ CC_n(x)\cap S=\emptyset] =$$
$$\underset{x\in U}{\mathbf{P}}[S \ \cap \ (CC_1(x) \ \cup \ CC_2(x) \ \cup \ \ldots \ \cup \ CC_n(x)) \ = \ \emptyset]$$

Proof. First we introduce the following property for sets. For any sets A, B and C,

$$A\cap(B\cup C)=\emptyset$$
$$(A\cap B)\cup(A\cap C)=\emptyset$$

Union of two sets is empty set if and only if both of the sets are empty. Therefore,

$$A\cap(B\cup C)=\emptyset \iff (A\cap B)=\emptyset\wedge(A\cap C)=\emptyset$$

Using this property, events in Eq. (4.4) can be shown to be equal as follows:

$$CC_1(x)\cap S=\emptyset \ \wedge \ CC_2(x)\cap S=\emptyset \ \wedge \ \ldots \ \wedge \ CC_n(x)\cap S=\emptyset$$
$$(CC_1(x)\cup CC_2(x))\cap S=\emptyset \ \wedge \ \ldots \ \wedge \ CC_n(x)\cap S=\emptyset$$
$$\vdots$$
$$S \ \cap \ (CC_1(x) \ \cup \ CC_2(x) \ \cup \ \ldots \ \cup \ CC_n(x)) \ = \ \emptyset$$

□

For the sake of simplicity following is defined:

$$\mathcal{C} \triangleq CC_1(x) \ \cup \ CC_2(x) \ \cup \ \ldots \ \cup \ CC_n(x) \tag{4.4.5}$$

Since all of the collision classes are identical in terms of shape and size, the expression in Eq. 4.4 can be restated as, what is the probability that S strings randomly selected out of 2^w strings are disjoint from $\mathcal{C}$. Assuming that S is **uniform** over U,

$$\underset{x \in U}{\mathrm{P}}[S \cap \mathcal{C} = \emptyset] = \frac{\binom{2^w - |\mathcal{C}|}{|\mathcal{S}|}}{\binom{2^w}{|S|}} \tag{4.4.6}$$

Here, $\binom{2^w}{|\mathcal{S}|}$ is the total number of possible selections of $|S|$ strings out of 2^w total number of strings where we are selecting $|S|$ strings out of $2^w - |\mathcal{C}|$ total number of strings. This is a formulation of unordered sampling without replacement [41], that is once a string is selected in $\mathcal{S}$ it cannot be selected again.

When $|\mathcal{S}| \ll 2^w$, Equation 4.4.6 can be simplified to

$$\underset{x \in U}{\mathrm{P}}[S \cap \mathcal{C} = \emptyset] = \left(\frac{2^w - |\mathcal{C}|}{2^w}\right)^{|S|} \tag{4.4.7}$$

Here, $|\mathcal{S}|$ independent selections are made repeatedly out of the subset with $2^w - |\mathcal{C}|$ strings. Since the size of collision class intersections can be computed with aforementioned properties, the size of set $\mathcal{C}$ can be computed using set inclusion & exclusion. Hitherto, everything required to compute the false positive probability of a BF without the assumptions of uniformity and independence is presented. The only assumption used in this method is that S is uniformly distributed over U.

4.5 Example Computation of P_{fp} for $k = 2$

$$1 - P_{fp} = \mathop{\mathbf{P}}_{x \in U}[CC_1(x) \cap S = \emptyset] + \mathop{\mathbf{P}}_{x \in U}[CC_2(x) \cap S = \emptyset]$$
$$- \mathop{\mathbf{P}}_{x \in U}[S \cap (CC_1(x) \cup CC_2(x)) = \emptyset]$$
$$\mathop{\mathbf{P}}_{x \in U}[CC_1(x) \cap S = \emptyset] = \frac{\binom{2^w - |CC_1(x)|}{|S|}}{\binom{2^w}{|S|}}$$
$$\mathop{\mathbf{P}}_{x \in U}[CC_2(x) \cap S = \emptyset] = \frac{\binom{2^w - |CC_2(x)|}{|S|}}{\binom{2^w}{|S|}}$$
$$\mathop{\mathbf{P}}_{x \in U}[S \cap (CC_1(x) \cup CC_2(x)) = \emptyset] = \frac{\binom{2^w - |CC_1(x) \cup CC_2(x)|}{|S|}}{\binom{2^w}{|S|}}$$
$$|CC_1(x)| = 2^{w - dim(H_1)} - 1$$
$$|CC_2(x)| = 2^{w - dim(H_2)} - 1$$
$$|CC_1(x) \cup CC_2(x)| = |CC_1(x)| + |CC_2(x)| - |CC_1(x) \cap CC_2(x)|$$
$$|CC_1(x) \cap CC_2(x)| = 2^{w - dim([H_1 H_2])} - 1$$

4.6 Pseudocode for the Novel False Positive Computation

Algorithm 1 False Positive Computation

Input: $hash_matrices[w][y][k]$, $|S|$

Output: P_{fp}

$1 - P_{fp} \leftarrow 0$
$w \leftarrow size(hash_matrices, 1)$
$y \leftarrow size(hash_matrices, 2)$
$k \leftarrow size(hash_matrices, 3)$
$hash_idx_list \leftarrow [1:1:k]$
for $i = 1 : k$ **do**
    $sign_pr \leftarrow (-1)^{i-1}$
    $combs_pr \leftarrow nchoosek(hash_idx_list, i)$
    for $j = 1 : size(combs_pr, 1)$ **do**
        $ec_union_size \leftarrow 0$
        for $t = 1 : size(combs_pr, 2)$ **do**
            $sign_ec \leftarrow (-1)^{t-1}$
            $combs_ec \leftarrow nchoosek(combs_pr(j, :), t)$
            for $v = 1 : size(combs_ec, 1)$ **do**
                $rank_of_col_union \leftarrow gfrank_hash_union(hash_matrices, combs_ec(v, :))$
                $ec_union_size \leftarrow ec_union_size + sign_ec \times 2^{w - rank_of_col_union}$
            end for
        end for
        $1 - P_{fp} \leftarrow 1 - P_{fp} + sign_pr \times intersect_prob_expr(ec_union_size, w, |S|)$
    end for
end for
$P_{fp} \leftarrow 1 - (1 - P_{fp})$

$nchoosek(v, k)$ is the MATLAB function that returns every possible k element combinations of elements in list v. Note that return value is a matrix where each combi-

nation is placed in a row.

$gfrank_hash_union(hash_matrices, idx_list)$ returns the dimension in GF(2) of the matrix obtained by horizontally concatenating hash matrices where hash indexes given in idx_list and $hash_matrices$ is a $w \times y \times k$ matrix containing binary hash matrices.

$intersect_prob_expr(ec_union_size, w, |S|)$ is the custom function that returns the numeric value for the expression given in either Equation 4.4.6 or 4.4.7.

4.7 Algorithmic Complexity

Algorithm 1, has two main time consuming parts. Namely, these are $gfrank_hash_union()$ and $intersect_prob_expr()$ where GF rank computation is found to be more computationally expensive by profiling the algorithm for various parameters.

Analyzing the ranges of nested for loops, one can see that, the number of times these subroutines are called is as follows:

- $gfrank_hash_union() : 3^k - 2^k$

- $intersect_prob_expr() : 2^k - 1$

As it turns out, the implementation in Algorithm 1 recomputes the rank of same matrices more than once. Alternating the implementation, in a way that rank of every combination of every size for all hash functions at the beginning once the function is called, one obtains the algorithm in Algorithm 2.

Algorithm 2 False Positive Computation with Precomputation of Ranks

Input: $hash_matrices[w][y][k]$, $|S|$

Output: P_{fp}

$1 - P_{fp} \leftarrow 0$

$w \leftarrow size(hash_matrices, 1)$

$y \leftarrow size(hash_matrices, 2)$

$k \leftarrow size(hash_matrices, 3)$

$hash_idx_list \leftarrow [1:1:k]$

for $hash_comb$ in all possible HF combinations of all sizes **do**

    $hash_rank_array(hash_comb) \leftarrow gfrank_hash_union(hash_matrices, combs_ec(v,:))$

end for

for $i = 1:k$ **do**

    $sign_pr \leftarrow (-1)^{i-1}$

    $combs_pr \leftarrow nchoosek(hash_idx_list, i)$

    for $j = 1:size(combs_pr, 1)$ **do**

        $ec_union_size \leftarrow 0$

        for $t = 1:size(combs_pr, 2)$ **do**

            $sign_ec \leftarrow (-1)^{t-1}$

            $combs_ec \leftarrow nchoosek(combs_pr(j,:), t)$

            for $v = 1:size(combs_ec, 1)$ **do**

                $rank_of_col_union \leftarrow hash_rank_array(combs_ec(v))$

                $ec_union_size \leftarrow ec_union_size + sign_ec \times 2^{w - rank_of_col_union}$

            end for

        end for

        $1 - P_{fp} \leftarrow 1 - P_{fp} + sign_pr \times intersect_prob_expr(ec_union_size, w, |S|)$

    end for

end for

$P_{fp} \leftarrow 1 - (1 - P_{fp})$

In the version of implementation given in Algorithm 2, the number of subroutine calls is as follows:

- $gfrank_hash_union() : 2^k - 1$
- $intersect_prob_expr() : 2^k - 1$

Note that the computational complexity of $gfrank_hash_union()$ is also dependent of the matrix size which in this case depends on k, y, w. However the exact complexity depends on the implementation which is beyond the scope of this work.

CHAPTER 5

EVALUATION

5.1 Hardware Simulation Environment

5.1.1 Problem Definition

In the previous section, we have come up with a new false positive probability expression for bloom filters with H3 hash functions that do not assume hash uniformity or independence, unlike the legacy expression.

Our new expression aims to produce correct false positive results in the presence of non-uniform, dependent hash functions. We expect that as the amount of dependence, and non-uniformity is increased, the actual false positive probability will deviate from the legacy expression output while our expression will still yield accurate results.

Therefore, we need a method to estimate/measure the false positive rate of a given bloom filter so that we can compare our expression with legacy expression with a reference.

Our initial attempt at this problem was to build a soft simulation environment where uniformly randomly generated members were programmed into a BF implemented on software and stored in memory for referencing. After the programming, randomly generated inputs are queried with BF for membership. During the simulation, the numbers of true positives, false positives and true negatives are counted and the observed false positive rate is computed.

The problem with this approach was that it was computationally expensive. Assuming that n queries are to be made on a BF with k hash functions, $\Theta(kn)$ hash computations are required. Depending on the application, the required false positive probability for a BF can be as small as 10^{-8}. Therefore, to estimate the false positive probability with significant confidence, the number of queries n required is found to be infeasible with this approach.

As a result, instead of performing these simulation runs on software, we employed a new approach in which we perform these experiments on an FPGA where bloom filter query complexity for an individual query is $\mathcal{O}(1)$ and run time is only determined by the number of queries. Since each query can be completed in a single clock cycle, around $5ns$, hardware simulation provides a throughput way beyond the range achievable on software.

In order to use this hardware simulation environment as a reference, we need to demonstrate that it operates properly. For this purpose, the hardware simulation environment is verified where legacy false positive expression is known to operate accurately, i.e. where its assumptions are met.

In the remainder of this section, we will introduce the hardware test bench architecture in detail. After that, we will discuss the implementation of this design. Finally, we will provide results that suggest the proper operation of this test suite.

5.1.2 Testbench Design

The hardware simulation design is composed of the following modules: Stimulus Generator, BF Under Test, and Scoreboard. The overall structure including submodule interconnections is given in Figure 5.1.

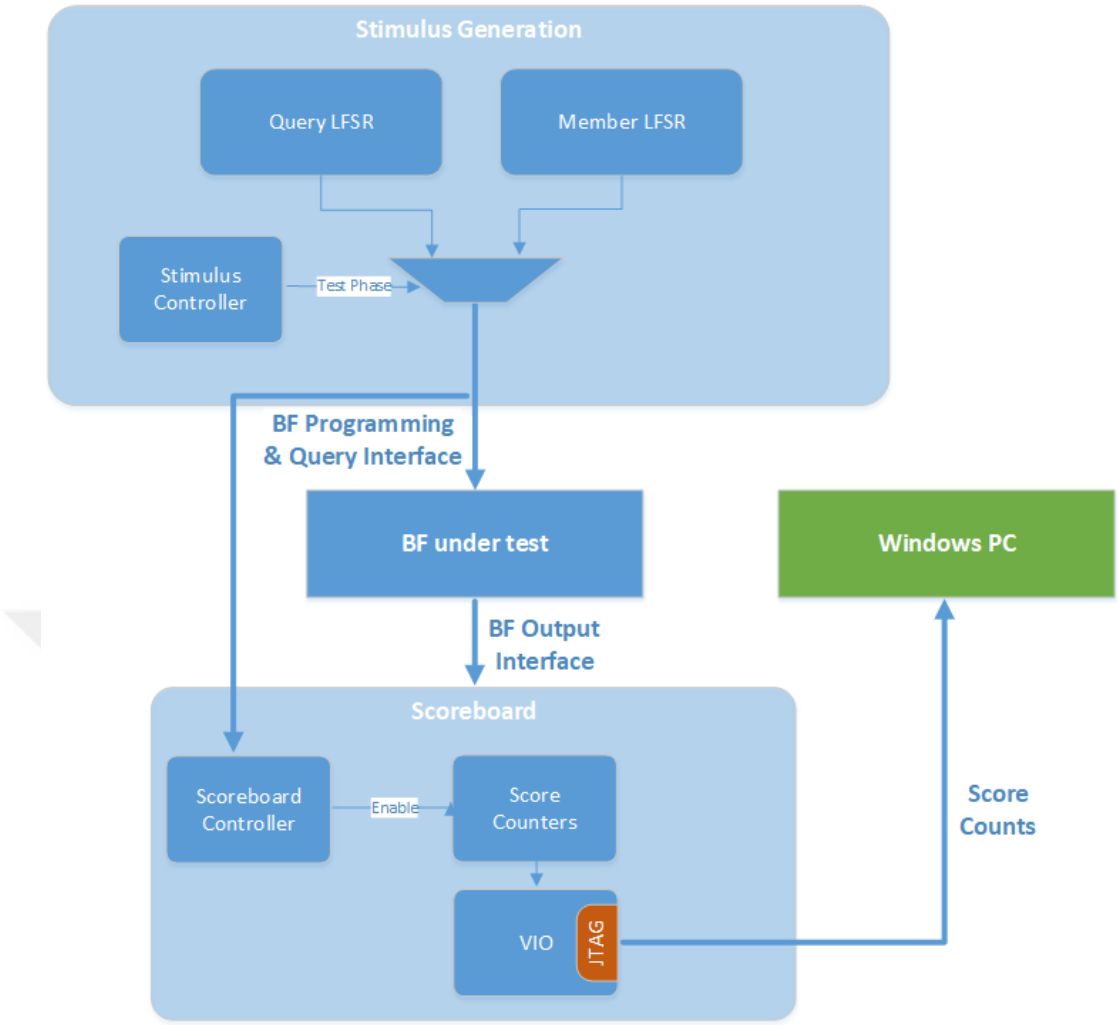


Figure 5.1: Block design of test bench.

Stimulus generation module contains the following modules:

- **Member LFSR (Linear Feedback Shift Register):** A w-bit wide LFSR to randomly generate the members, $x \in \mathcal{S}$, where w is the size of BF inputs.
- **Query LFSR:** Another LFSR with rank $r > w$, that randomly generates inputs for queries.
- **Stimulus Controller:** A stimulus controller that drives the BF inputs during programming and query phases.

To this end

At the beginning of the simulation, Stimulus Controller uses Member LFSR to generate the member set $\mathcal{S}$ to be programmed into the BF. The characteristic polynomial of Member LFSR is a prime polynomial, and LFSR is initiated with a random non-zero seed value. In addition, the size of the LFSR is w bits. Since a prime polynomial of size w produces the same output only after $2^w - 1$ iterations, members generated by Member LFSR are uniform and distinct.

After the programming phase, Stimulus Controller begins to use Query LFSR to generate random inputs to BF under test. The characteristic polynomial of Query LFSR is also a prime and has a rank r, larger than w. Query LFSR is also seeded with a random non-zero seed. The output of Query LFSR is a sequence that repeats itself in $2^r - 1$ iterations. BF input is taken from a w bit portion of the LFSR output. In each cycle of LFSR output sequence, each potential non-zero value of w bits portion is observed 2^{r-w} times and 0 is observed $2^{r-w} - 1$ times. Hence, it is safe to say that the queried inputs are uniform by design.

BF under test is a bloom filter implementation with partitioned memory. The implemented bloom filter has an interface that provides programming and query functionalities. Query outputs are produced once every clock cycle with a latency depending on the amount of pipelining.

Scoreboard module contains a scoreboard controller that monitors the BF programming interface and enables the score counters when the programming phase is over. Score counters count the number of times the bloom filter generates positive and negative outputs without considering the inputs. The values in score counters are retrieved to test PC through a virtual I/O module with JTAG interface.

At the end of each simulation run following raw results are obtained:

- $\#\ of\ positive\ outputs$
- $\#\ of\ negative\ outputs$

From these results, the total number of queries can be easily computed,

$$total\ \#\ of\ queries = \#\ of\ positive\ outputs + \#\ of\ negative\ outputs$$

In order to distinguish between true and false positives, either there needs to be an exact matching mechanism or the stimulus needs to be produced in a manner that considers member and non-member strings which at least requires the storage of the member set. Instead of such resource and/or memory intensive approaches, the following approximation is used to estimate the false positive rate resulting from the simulation.

Since the queried inputs are uniformly generated, the probability that an input is actually a member programmed into BF under test is as follows:

$$P_{member} = \frac{|S|}{2^w}$$

As a result, the expected number of queried inputs which are also a member is as follows:

$$\mathbb{E}\{\#\ of\ queried\ members\} = P_{member} \times total\ \#\ of\ queries$$

BF false positives are defined as positive outputs when the queried inputs are not members. Therefore, the false positive probability can be computed from the raw simulation results as follows:

$$P_{fp,test} = \frac{\#\ of\ positive\ outputs - \mathbb{E}\{\#\ of\ queried\ members\}}{total\ \#\ of\ queries - \mathbb{E}\{\#\ of\ queried\ members\}}$$

5.1.3 Implementation

Modules for this test suite are implemented on VHDL. Required inputs such as LFSR polynomials, LFSR seed values, and BF parameters are implemented as VHDL packages.

For each test, VHDL packages with configuration parameters and required TCL scripts are generated in Matlab. Generated TCL scripts perform the following operations:

1. Create a Vivado project and add all of the design files to the project.
2. Synthesize and implement the project.
3. Check the result of static timing analysis and ensure that there are paths with timing errors.
4. Program the hardware device and collect test results through virtual I/O.

ion module The connection between the test PC and the evaluation kit is made with a USB Type A to USB Micro B with the help of USB-JTAG converter residing on the evaluation kit.

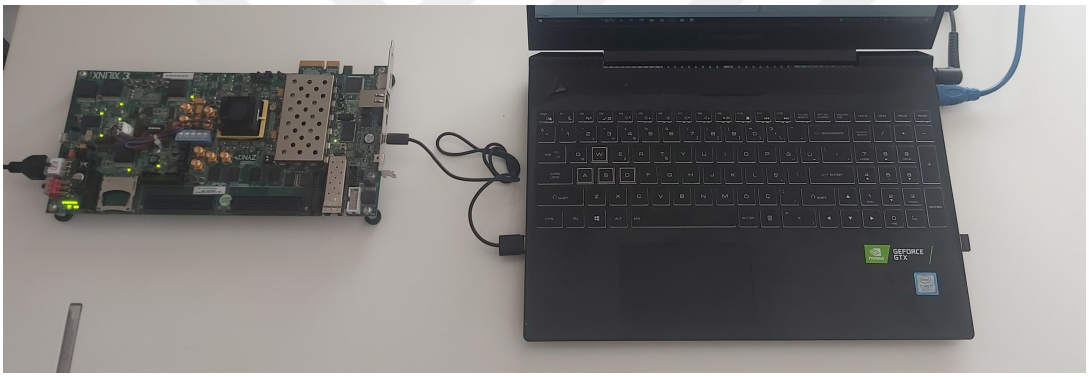


Figure 5.2: Photograph of the test setup.

5.1.4 Evaluation of Testbench

In order to verify our test suite, we conducted tests on a set of bloom filters where the assumptions made for the legacy expression can be met. To this end, we expect the measured Pfp to be the same as the result of the legacy Pfp. The test set is created as follows:

1. Initial test set is created with all combinations in following dimesions:
   - $k = 2, 4, \ldots, 18$

- $w = 32, 64, 200$
- $y = 15, 16, \ldots, 30$
- $S = 10^3, 10^4, 10^5, 2 \times 10^5$

where k is the number of hash functions, w is BF input width, y is hash output size, S is the size of the member set.

2. Eliminate all candidates that do not satisfy $w \geq ky$, which is a hard constraint for hash independence for BF with H3 hash functions.

3. Compute l, the number of BRAM blocks that will be addressed by each of the hash functions. Considering that the address width of an individual BRAM block is 15 in the target FPGA, the number of BRAMs for each hash function is computed as $l = log_2(y - 15)$.

4. Eliminate all candidates that do not satisfy $kl \leq 540$, which states that the total number of BRAMs should be smaller than what FPGA offers, which is, in this case, 540.

5. Eliminate all candidates that do not have a legacy false positive probability between 10^{-6} and 0.15. The lower bound is determined considering the feasible run times for tests with enough confidence in the estimated false positive rate. The upper bound is determined by the practicality of a BF by means of commonly observed applications.

The final size of the test set was found to be 93. For each BF in this set, a hardware test is conducted. Each test is performed for approximately 60 seconds where 2×10^8 queries are performed each second. During each test, Score Counts are logged for 6 times once every 10 second.

After completing the test, the false positive rate, $P_{fp,test}$ is computed as explained in Section 5.1.2. In order to show that the number of queries applied is sufficient, confidence interval deltas over 6 measurements of false positive rates for each test point is calculated as follows:

$$\delta = Z_{95}\frac{s}{\sqrt{n}}$$

where Z_{95}, s, and n are Z-score, the standard deviation of computed false positive rates, and the number of tests conducted which is 6 in this case. For each test point, δ is found to be always compusmaller than 1% of the mean false positive rate. This suggests that the number of queries applied to each test point is large enough to yield consistent results.

For each test point, percentage error (e_p) with respect to the legacy false positive probability $(P_{fp,leg})$ is computed as follows:

$$e_p = \frac{P_{fp,test} - P_{fp,leg}}{P_{fp,leg}} \times 100$$

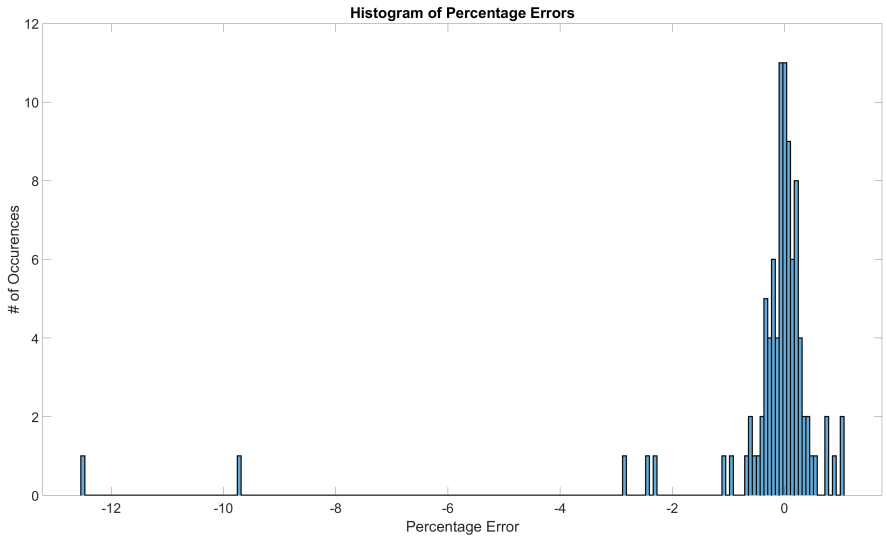


Figure 5.3: Histogram of percentage errors.

Histogram of the percentage errors can be seen in Figure 5.3. It can be seen that in only 2 of the test points, the absolute value of percentage error is larger than 5. The mean value of the absolute percentage error is found to be 0.56. In addition, some of the percentile values for the absolute value of percentage errors can be found in Table 5.1.

Table 5.1: Percentiles for absolute value of e_p.

Percentile	e_p
50	0.193
60	0.256
70	0.334
80	0.461
90	0.918
95	2.11

As can be seen from Table 5.1, in 80% of test points, the error between simulated and computed false positive rates is smaller than 0.461%. In a similar manner, in 95% of test points, the absolute value of percentage error is smaller than 2.11%.

In only two of the test points, the percentage error is found to be larger than 5%. The likely cause of this is that the test is conducted for one member set and one hash function which are selected randomly and can affect the results. These results are presented to demonstrate that the hardware test environment behaves as expected, and the results strongly suggest this point.

5.2 Evaluation Results

The legacy expression provides an estimate on false positive probability under the assumptions of uniform and independent hash functions. However what happens when these assumptions are violated is a question that cannot be answered by the legacy expression. This work presents an attempt to analyze and analytically express the behaviour of Bloom Filters when these assumptions are violated.

For this purpose, three experiments are designed and performed. The parameters of these three experiments are $w = 64$, $k = 2, 3$ and $y = 20$. Here we note that $w > k \cdot y$ for these experiments which allows all hash functions to be uniform and independent.

We select the numerical values for these parameters to be in the range of the appli-

cations presented in the previous work of hardware BFs with H_3 hash functions as in [13, 12, 11].

5.2.1 Experiment 1

First experiment is constructed with the following parameters:

- $w = 64$
- $k = 2$
- $y = 20$
- $|\mathcal{S}| = 10539$

When computed with the legacy expression, this bloom filter has a false positive probability of approximately 10^{-4}. In the initial design points all dependencies are equal to zero. From that point on, test points are constructed in two paths. These paths are as follows:

1. In the first path, test points are constructed by increasing $dep(H_2)$ one by one while maintaining $dep(H_1)$ as zero. Note that due to P9.3, $dep(H_1, H_2)$ also increases as we increase $dep(H_2)$.
2. In the second path, test points are constructed by increasing only $dep(H_1, H_2)$ while both $dep(H_1)$ and $dep(H_2)$ remains as zero.

In the first path, since the dimension of a matrix is decreased from y, we gradually lose uniformity of H_2 and independence of H_1, H_2. On the other hand, in the second path both hash functions are uniform, however in each step hash functions become more and more dependent.

The result of the experiments are presented in Figures 5.4 and 5.5. Figure 5.4 presents the legacy false positive probability $P_{fp,leg}$, false positive probability computed with our expression P_{fp} and false positive rates obtained from hardware simulations $P_{fp,test}$

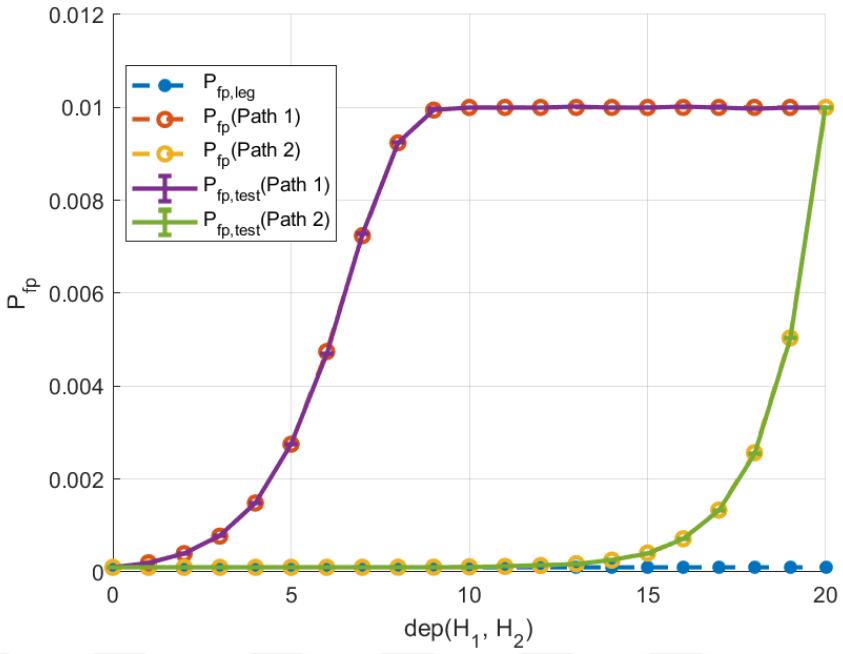


Figure 5.4: False positive rates obtained from legacy expression, our expression and tests with respect to $dep(H_1, H_2)$

.

for path 1 and 2. Figure 5.5 present the same information divided by the constant legacy value.

It can be seen that when dependencies are not present, i.e. $dep(H_1, H_2) = 0$, both P_{fp} and $P_{fp,test}$ complies with the legacy expression. In Path 1, where uniformity is gradually disrupted, when $dep(H_2)$ is increased from 0 to 1, actual false positive rate is doubled compared to the one computed with the legacy expression. In addition, as $dep(H_2)$ exceeds 10, actual false positive rate converges to approximately 0.01. This behaviour can explained by legacy false positive probability computed for a BF with $k = 1$ and where all of the other design parameters are the same. This false positive probability is approximately 0.0095, which suggests that as $dep(H_2)$ is increased beyond a point, H_2 becomes useless compared to H_1.

In Path 2, as $dep(H_1, H_2)$ is increased both P_{fp} and $P_{fp,test}$ diverge from the legacy

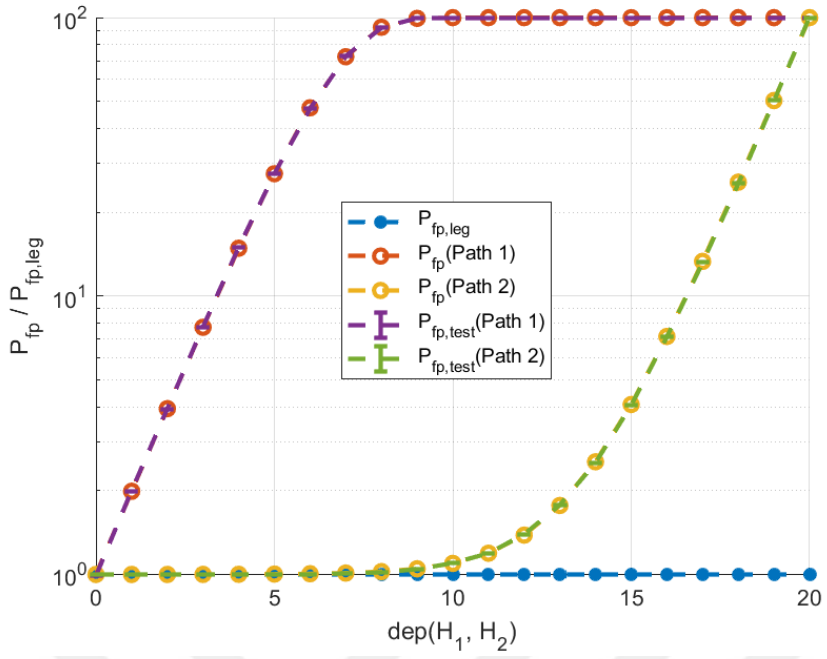


Figure 5.5: False positive rate ratios obtained from legacy expression, our expression and tests with respect to $dep(H_1, H_2)$

value. Compared to Path 1, the effect of increasing $dep(H_1, H_2)$ becomes significant later. This suggests that the loss of uniformity can have a more adverse effect on performance compared to the loss of independence. As $dep(H_1, H_2)$ becomes 20, the false positive rate again reaches 0.01, which makes sense since both hash functions are totally dependent and they effectively behave as only one hash function.

So why does increasing $dep(H_2)$ has more adverse effect than increasing $dep(H_1, H_2)$? The answer to this question again lies in the the size of collision classes and their intersections. As we increase $dep(H_2)$, the size of the collision class for H_2 becomes exponentially larger which increases the probability of collision for that hash function. In addition, increasing $dep(H_2)$ also increases $dep(H_1, H_2)$ which means the size of the collision class intersection also gets larger. On the other hand, in Path 2, we only increase $dep(H_1, H_2)$ while the size of individual collision classes remains the same. Note that, when $dep(H_1H_2)$ reaches 20, sizes of $CC_1(x)$, $CC_2(x)$ and $CC_1(x) \cap CC_2(x)$ becomes identical, which indicates that collision classes are actu-

ally one and the same. Hence, when $dep(H_1, H_2) = 20$, BF behaves as if $k = 1$.

5.2.2 Experiment 2

In the second experiment, we work on BFs with following parameters:

- $w = 64$
- $k = 3$
- $y = 20$
- $|\mathcal{S}| = 49836$

As in Experiment 1, member set size is deliberately selected to yield a legacy false positive probability of approximately 10^{-4}.

In the initial design points all dependencies are equal to zero. From that point on, test points are constructed in three paths. These paths are as follows:

1. In the first path, test points are constructed by increasing $dep(H_3)$ one by one while maintaining $dep(H_1)$, $dep(H_2)$ as zero, while all other dependencies are ensured to be their corresponding minimum. For instance, $dep(H_1, H_3)$ also increases by one as we increase $dep(H_3)$ by one.
2. In the second path, test points are constructed by increasing $dep(H_2, H_3)$ while both $dep(H_1)$, $dep(H_2)$ and $dep(H_3)$ remains as zero and other dependencies are kept at their minimum.
3. In the third path, test points are constructed by increasing only $dep(H_1, H_2, H_3)$ while all other dependencies are equal to zero.

As can be seen in Figures 5.6 and 5.7, loss of uniformity in Path 1 and increased dependency between two matrices in Path 2 behave very similarly to Experiment 1.

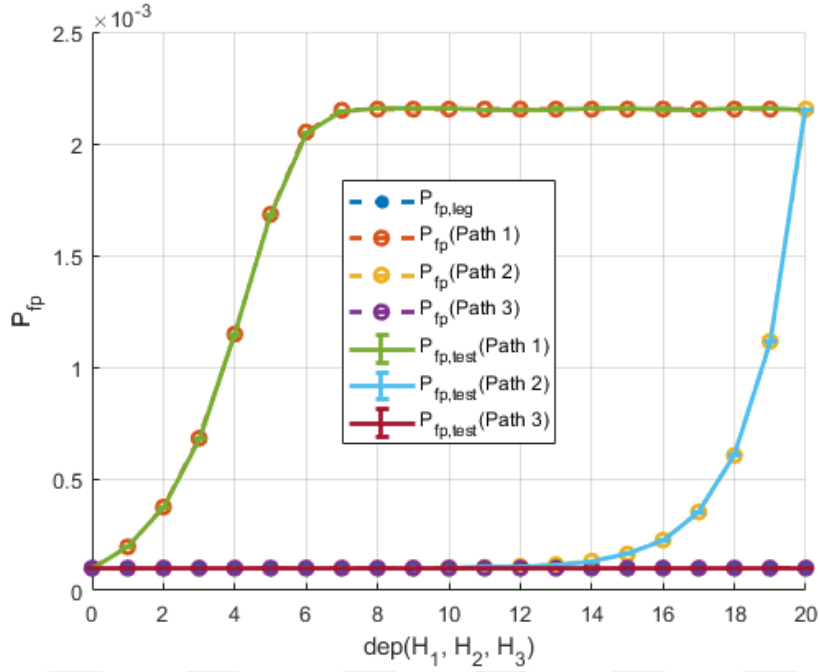


Figure 5.6: False positive rates obtained from legacy expression, our expression and tests with respect to $dep(H_1, H_2, H_3)$

Comparing Paths 2 and 3, one can see that when $dep(H_1, H_2, H_3)$ is same increased dependency among two hash functions have a more significant effect compared to the increased rank among three hash functions. This is again caused by P9.3. When $dep(H_1, H_2)$ is increased by one, $dep(H_1, H_2, H_3)$ increases at least by one. As a result, when dependencies are not avoidable, for instance when $w < ky$ it is preferable to have them only among large number of hash functions.

As can be seen in Experiments 1 and 2, as the dependencies are increased actual false positive rate of the Bloom Filter deviates from the one computed by the legacy expression. In addition, these experiments demonstrate that our false positive probability expression can yield accurate results in the presence of dependencies.

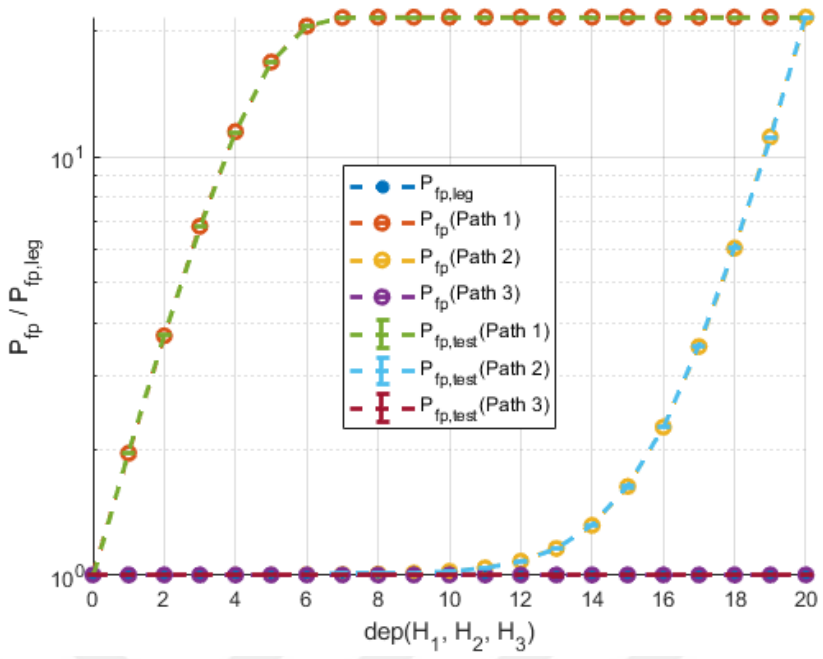


Figure 5.7: False positive rate ratios obtained from legacy expression, our expression and tests with respect to $dep(H_1, H_2, H_3)$

.

5.2.3 Experiment 3

In this experiment, we try to answer the following question: Is it enough to only check $dep(H_1, H_2, H_3)$ to estimate the effect of dependencies?

Table 5.2: Test points and results for Experiment 3.

w	k	y	S	$dep(H_1)$	$dep(H_2)$	$dep(H_3)$	$P_{fp,leg}$	P_{fp}	$P_{fp,test}$
64	3	20	49836	0	0	6	10^{-4}	0.0021	0.0020
64	3	20	49836	0	3	3	10^{-4}	0.0046	0.0047
64	3	20	49836	2	2	2	10^{-4}	0.0052	0.0052

From Table 5.2, one can see that the answer to this question is negative. In all the experiments, $dep(H_1, H_2, H_3) = 6$. However the amount of deviation from the legacy false positive probability is significant.

5.2.4 Experiment 4

In this experiment, we aim to analyze the effect of pairwise dependency, e.g. $dep(H_1, H_2)$, for BFs with different k.

The BF parameters are selected as follows:

- $w = 256$
- $y = 20$
- $k = 2 \dots 10$
- $|\mathcal{S}|$ is selected to yield $P_{fp,legacy} = 10^{-4}$ for each BF.

Notice that since $w = 256$, $y = 20$ and maximum value of $k = 10$, $w > ky$ is always satisfied. For all BFs with different number of hash functions, base designs are created with all dependencies equal to zero. Then dependencies are introduced only between hash pairs h_1 and h_2 such that $dep(H_1, H_2)$ is increased to 10, 15 and 20. Note that during this operation all of the other dependencies are maintained at their corresponding minimum, e.g. $dep(H_1)) = dep(H_2) = 0$, $dep(H_1, H_2, H_3) = dep(H_1, H_2)$, $dep(H_4, H_5, H_6, H_7) = 0$ etc.

For all the BFs when dependencies are zero, $P_{fp}/P_{fp,legacy} = 1.0$. This indicates that legacy computation and our expression yield identical results as expected.

The rest of the results are presented in Table 5.3. For all $dep(H_1, H_2)$, it can be seen that as k increases the effect of single pair dependency over BF performance gets smaller. For instance, when $dep(H_1, H_2) = 10$, change in actual false positive rate is found to be 9.6% for $k = 2$ but as k exceeds 6, the increase in the false positive rate becomes less than 0.5%.

In addition, for all BFs with different k, as the dependency between h_1 and h_2,

$dep(H_1, H_2)$ increases, $P_{fp}/P_{fp,legacy}$ becomes larger as expected.

Table 5.3: Ratio of $P_{fp}/P_{fp,legacy}$ for $k = 2..10$, $w = 256$ when $dep(H_1, H_2)$ is 0, 10, 15 and 20.

		$P_{fp}/P_{fp,leg}$		
k	$\|\mathcal{S}\|$	$dep(H_1, H_2) = 10$	$dep(H_1, H_2) = 15$	$dep(H_1, H_2) = 20$
2	10539	1.096	4.078	100
3	49837	1.019	1.627	21.54
4	110479	1.008	1.267	10
5	180939	1.005	1.152	6.3
6	254423	1.003	1.1	4.6
7	327516	1.002	1.073	3.7
8	398596	1.0017	1.056	3.2
9	466953	1.0014	1.045	2.8
10	532337	1.0011	1.037	2.5

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this work, we focus on the shortcomings of the legacy false positive expression for Bloom Filters that use H3 hash functions. In this sense, we have used GF(2) vector algebra to derive properties for H3 BFs. These properties are then used to define a measure of dependency on and among the hash functions of BFs. Moreover, we present a novel method to estimate the false positive probability accurately when assumptions made by legacy expression is violated such as uniformity and independence.

We implement our method and compared its accuracy with the results of legacy expression for a variety of structured test points. Furthermore, we design and implement a test suite on actual hardware and use it as a reference for the aforementioned comparisons between our method and legacy expression. We observe that when the assumptions of legacy false positive expression are violated, the actual false positive probability may be quite different from the one computed with the legacy expression. In all cases presented, our method yields results that comply with the ones obtained from the tests.

Furthermore, we observe that for a BF with k hash functions, loss of uniformity is the strongest type of dependency in terms of impact on false positive probability. Dependencies among pairs of hash functions have weaker effects on results compared to the loss of uniformity. Dependencies among three and more hash functions are found to be insignificant. Moreover, as k increases the effect of dependencies becomes weaker. To sum up, it is presented that the proposed framework and resulting false positive

rate estimation method can provide one the means to analyze H_3 BF in the absence of uniformity and independence.

As a future work, we are planning to use GF(2) vector algebra on H3 BFs to create collision free hash functions and false positive free BFs for a given member set.

REFERENCES

[1] S. Zengin and E. G. Schmidt, "A fast and accurate hardware string matching module with bloom filters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 305–317, 2016.

[2] S. Dharmapurikar and J. W. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems," *IEEE Journal on Selected Areas in communications*, vol. 24, no. 10, pp. 1781–1792, 2006.

[3] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM transactions on networking*, vol. 8, no. 3, pp. 281–293, 2000.

[4] F. Hao, M. Kodialam, T. Lakshman, and H. Song, "Fast dynamic multiple-set membership testing using combinatorial bloom filters," *IEEE/ACM Transactions On Networking*, vol. 20, no. 1, pp. 295–304, 2011.

[5] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[6] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," *IEEE/ACM Transactions on Networking*, vol. 14, no. 2, pp. 397–409, 2006.

[7] H. Lim, K. Lim, N. Lee, and K.-H. Park, "On adding bloom filters to longest prefix matching algorithms," *IEEE Transactions on Computers*, vol. 63, no. 2, pp. 411–423, 2012.

[8] Y. Wu, J. He, S. Yan, J. Wu, T. Yang, O. Ruas, G. Zhang, and B. Cui, "Elastic bloom filter: Deletable and expandablefilter using elastic fingerprints," *IEEE Transactions on Computers*, 2021.

[9] J. H. Mun and H. Lim, "New approach for efficient ip address lookup using a

bloom filter in trie-based algorithms," *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1558–1565, 2015.

[10] Y. Zhu, H. Jiang, J. Wang, and F. Xian, "Hba: Distributed metadata management for large cluster-based storage systems," *IEEE transactions on parallel and distributed systems*, vol. 19, no. 6, pp. 750–763, 2008.

[11] R. Quislant, E. Gutierrez, O. Plata, and E. L. Zapata, "Ls-sig: Locality-sensitive signatures for transactional memory," *IEEE Transactions on Computers*, vol. 62, no. 2, pp. 322–335, 2011.

[12] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, "Implementing signatures for transactional memory," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 123–133, IEEE, 2007.

[13] S. Pontarelli and M. Ottavi, "Error detection and correction in content addressable memories by using bloom filters," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1111–1126, 2012.

[14] H. Yang, Y. Liang, J. Yuan, Q. Yao, A. Yu, and J. Zhang, "Distributed blockchain-based trusted multidomain collaboration for mobile edge computing in 5g and beyond," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 11, pp. 7094–7104, 2020.

[15] Y.-L. Chen, B.-Y. Chang, C.-H. Yang, and T.-D. Chiueh, "A high-throughput fpga accelerator for short-read mapping of the whole human genome," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 6, pp. 1465–1478, 2021.

[16] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[17] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2011.

[18] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1378–1381, 1997.

[19] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," in *11th Symposium on High Performance Interconnects, 2003. Proceedings.*, pp. 44–51, IEEE, 2003.

[20] R. Rivest, "The md5 message-digest algorithm," tech. rep., 1992.

[21] H. Vandierendonck and K. De Bosschere, "Xor-based hash functions," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 800–812, 2005.

[22] "Snort - network intrusion detection & prevention system." `https://www.snort.org/`. (Accessed on 07/22/2022).

[23] S. Pei, K. Xie, X. Wang, G. Xie, K. Li, W. Li, Y. Li, and J. Wen, "Bhbf: A bloom filter using b h sequences for multi-set membership query," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 16, no. 5, pp. 1–26, 2022.

[24] "The caida anonymized internet traces dataset (april 2008 - january 2019) - caida." `https://www.caida.org/catalog/datasets/passive_dataset/`. (Accessed on 05/08/2022).

[25] "Geoff huston - potaroo.net." `https://www.potaroo.net/`.

[26] Y.-H. Lin, W.-C. Shih, and Y.-K. Chang, "Efficient hierarchical hash tree for openflow packet classification with fast updates on gpus," *Journal of Parallel and Distributed Computing*, vol. 167, pp. 136–147, 2022.

[27] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[28] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *2008 IEEE International Symposium on Workload Characterization*, pp. 35–46, IEEE, 2008.

[29] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," *Random Structures & Algorithms*, vol. 33, no. 2, pp. 187–218, 2008.

[30] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[31] J. L. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[32] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pp. 75–88, 2014.

[33] "Hp labs : Cacti." `https://www.hpl.hp.com/research/cacti/`. (Accessed on 05/06/2022).

[34] G. Cheng, D. Guo, L. Luo, J. Xia, and S. Gu, "Lofs: A lightweight online file storage strategy for effective data deduplication at network edge," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, pp. 2263–2276, 2021.

[35] D. Eastlake 3rd and P. Jones, "Us secure hash algorithm 1 (sha1)," tech. rep., 2001.

[36] N. Dutta, "An approach for fib construction and interest packet forwarding in information centric network," *Future Generation Computer Systems*, vol. 130, pp. 269–278, 2022.

[37] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *European Symposium on algorithms*, pp. 684–695, Springer, 2006.

[38] "ndnsim 2.0 documentation — overall ndnsim 2.0 documentation." `https://ndnsim.net/2.0/`. (Accessed on 07/25/2022).

[39] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of computer and system sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[40] R. B. Allenby and A. Slomson, *How to count: An introduction to combinatorics*. Chapman and Hall/CRC, 2010.

[41] H. Pishro-Nik, “Introduction to probability, statistics, and random processes,” 2016.