Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia

COPPE UFRJ

ON OPTIMIZATION OF HARDWARE-ASSISTED SECURITY

Leandro Santiago de Araújo

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Sandip Kundu
Leandro Augusto Justen Marzulo

Rio de Janeiro
Julho de 2019

ON OPTIMIZATION OF HARDWARE-ASSISTED SECURITY

Leandro Santiago de Araújo

RIO DE JANEIRO, RJ – BRASIL
JULHO DE 2019

Araújo, Leandro Santiago de

On Optimization of Hardware-Assisted Security/Leandro Santiago de Araújo. – Rio de Janeiro: UFRJ/COPPE, 2019.

XVIII, 174 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França
Sandip Kundu
Leandro Augusto Justen Marzulo

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2019.

Referências Bibliográficas: p. 157 – 174.

1. *PUF*. 2. Taint Analysis. 3. DIFT. 4. Weightless Neural Network. 5. Bloom Filter. 6. Cuckoo Filter. 7. Quotient Filter. 8. Approximate Membership Query. 9. Multi-index Hashing. 10. Hardware Security. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Meu filho, guarda a sabedoria e a reflexão, não as percas de vista. Elas serão a vida de tua alma e um adorno para teu pescoço. Então caminharás com segurança, sem que o teu pé tropece. Se te deitares, não terás medo. Uma vez deitado, teu sono será doce. Não terás a recear nem terrores repentinos, nem a tempestade que cai sobre os ímpios. Porque o Senhor é tua segurança e preservará teu pé de toda cilada.
(Bíblia, Provérbios 3:21-26)

Dedico este trabalho à minha namorada Isis.

Agradecimentos

Primeiramente, agradeço à minha namorada Isis, que me aguentou durante todo este tempo e que me apoiou em todas as minhas decisões, mesmo sabendo o quão difícil seria suportar a distância, e por ter me ajudado a manter o foco e me mostrar que sempre há uma luz no fim do túnel. Agradeço pelo seu companherismo, sua paciência e cumplicidade. Obrigado por tudo que tem feito por mim e por acreditar em mim. Te amo!

Agradeço à minha mãe, por me dá forças para seguir em frente e ter investido em mim, me ensinado o grande valor do estudo, educação e respeito.

Agradeço ao meus orientadores Felipe França e Leandro Marzulo que me acompanharam durante o mestrado e o doutorado. Obrigado pelos incetivos, confiança e por todos os ensinamentos. Esse trabalho só foi possível por vocês acreditarem na minha capacidade. Também agradeço por me disponibilizarem uma das bolsas de doutorado sanduíche vinculados ao projeto em parceria com a UMass. Sou muito grato por todas as oportunidades que vocês me deram.

Agradeço ao meu orientador Sandip Kundu, que começou a me orientar no período de doutorado sanduíche na UMass. Obrigado por me guiar e motivar a conhecer diversas áreas de pesquisas relacionadas à segurança. Grande parte desse trabalho é fruto das contribuições realizadas durante minha estadia na UMass. Também cito o professor Israel Koren, no qual contribuiu com o desenvolvimento do meu trabalho.

Agradeço aos amigos que fiz durante o doutorado sanduíche, dentre eles Nur, Felipe, Priscilla, Brenno, Camila, Maurício, Juliana, que me acolheram e me ajudaram durante todo o período fora do Brasil. Sou muito feliz por Deus ter colocado vocês na minha vida. Também agradeço aos colegas de laboratório da UMass, em especial ao Vinay Patil pelas parcerias em vários projetos e pelas conversas técnicas que foram de grande aprendizado para mim.

Agradeço aos amigos da UFRJ, dentre eles Brunno, Victor Cruz, Rui, entre outros. Obrigado pelas conversas descontraídas, por sempre estarem disponíveis em situações complicadas e pelas motivações.

À todos que contribuem em fazer Programa de Engenharia de Sistemas e Computação (PESC/COPPE/UFRJ) o melhor programa de pós-graduação do país.

Agradeço aos professores e aos funcionários por serem atenciosos e prestativos contribuindo com a minha formação acadêmica.

Às agências de fomento CNPq, CAPES e COPPETEC, pelas bolsas que permitiram custear minhas despesas durante período de doutorado sanduíche na UMass, assim como o período de doutorado no Brasil.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

SOBRE A OTIMIZAÇÃO DA SEGURANÇA ASSISTIDA POR HARDWARE

Leandro Santiago de Araújo

Julho/2019

Orientadores: Felipe Maia Galvão França
Sandip Kundu
Leandro Augusto Justen Marzulo

Programa: Engenharia de Sistemas e Computação

Physically Unclonable Functions (PUFs) surgiram como simples primitivas de segurança de hardware para implementar recursos de autenticação e geração de chaves criptográficas em dispositivos eletrônicos. Um Strong PUF ideal não é clonável e mapeia unicamente uma entrada de n-bits para uma saída de m-bits. Contudo, implementações reais de Strong PUFs possuem problemas de segurança. Esta tese propõe diversos modelos originais de Strong PUF, baseados na arquitetura de Redes Neurais sem Peso (RNP), resistentes contra ataques de construção de modelos por meio de algoritmos de aprendizado de máquina. A fabricação de grande volume de PUFs necessitam de técnicas de testes online para garantir a propriedade de exclusividade entre os PUFs fabricados. Uma solução de teste de PUF online baseado em Multi-Index Hashing (MIH) é otimizada através de estratégias de busca de similaridades para reduzir os recursos de memória. Dynamic Information Flow Tracking (DIFT) tem sido utilizado com sucesso para detectar acesso ilegal a informações confidenciais em tempo de execução. Nesta tese, um rastreador de fluxo implícito aninhado portátil é proposto para permitir que mecanismos baseados em fluxo explícito possam rastrear fluxos implícitos, inclusive em casos de aninhamento de laços profundo. Além do mais, uma nova regra de propagação é definida para mitigar a propagação incorreta dos dados afetados pela dependência de controle. Enfim, novos modelos de RNP baseados em estruturas de dados probabilísticas são propostas e analisadas com o objetivo de reduzir os requisitos de memória. Os novos modelos são robustos e são adequados como componentes para soluções de segurança assistidas por hardware.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

ON OPTIMIZATION OF HARDWARE-ASSISTED SECURITY

Leandro Santiago de Araújo

July/2019

Advisors: Felipe Maia Galvão França
Sandip Kundu
Leandro Augusto Justen Marzulo

Department: Systems Engineering and Computer Science

Physically Unclonable Functions (PUFs) have emerged as lightweight hardware security primitives to implement authentication and key generation features on electronic devices. An ideal Strong PUF cannot be cloned and maps an n-bit input to a unique m-bit output. However, real Strong PUF implementations suffer from security issues. This thesis proposes various novel Strong PUF designs, based on Weightless Neural Network (WNN) architecture, which are resistant against model building attacks through machine learning algorithms. Then, the proposed WNN PUFs are combined with a reliable entropy source to extend the reliability property to the final Strong PUF. High volume manufacturing of PUFs requires online testing techniques to ensure the desired uniqueness property among the manufactured PUFs. An online testing PUF solution based on Multi-Index Hashing (MIH) is optimized by similarity search strategies to reduce the memory resources. Dynamic Information Flow Tracking (DIFT) has been successfully utilized to detect illegal access to sensitive information at runtime. Nonetheless, recent evasion attacks explore implicit flows based on control dependencies that are not detectable by most of DIFT implementations, which only track data dependency propagation. In this thesis, a portable nested implicit flow tracking is proposed to enable explicit-flow based DIFT mechanisms track implicit flows, including deeply-nested branch scenarios. In addition, a new propagation rule is defined to mitigate the incorrect propagation of data under control dependencies. Finally, new WNN models based on probabilistic data structures are proposed and analyzed in order to reduce the memory requirements. The new models are robust and are suitable as components for hardware-assisted security solutions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The modern society is living the era of *ubiquitous computing* which is increasingly dependent on electronic devices in various areas such as banking, healthcare, autonomous car, smart homes, smart phones, supply chain and transportation. Progressively, more sensitive information are carried in these devices making security more expensive in terms of financial losses, safety and loss of privacy. Noteworthy examples are *Internet of Things* (IoT) architectures, including from traditional models where a central repository, such as a *Cloud* host, processes data collected by sensor to emergent solutions involving *Edge Computing* field where the data processing is distributed in end-devices. In both cases, data is exposed to integrity or confidentiality attacks during processing and the user has no control over it. The solely software-based security is not sufficient to protect those systems as advanced attacks can modify and corrupt them by exploring bugs on user software. Additionally, hardware support is required since the performance overhead from software-based security is non-negligible in some cases.

There are four fundamental concepts of information security to protect the storage and communication information:

1. **Confidentially** prevents unauthorized entities to access privileged information. Typically, it is ensured through encryption and decryption process where authorized entities have to use private keys to access the information.

2. **Integrity** verifies the completeness of data which can be corrupted when it is stored locally or used in a communication channel. A checksum is usually applied to provide this security.

3. **Authenticity** ensures the identity of the subject that requests or sends information. It relies on integrity in order to avoid any tampering information is undetected. The authenticity can be realized by some known information (e.g. a password), some restricted object (e.g. a smartcard) or some internal

characteristic (e.g. biometrics). Particularly, *Physical Unclonable Function* (PUF) is the latter category of authenticity as it uses physical fingerprint.

4. **Availability** allows the information is always available to be accessed. Communication channels and information systems are targeted from random faults and common attacks like denial-of-service attacks.

Over the last decade, many researches proposed *hardware-assisted security* solutions (also known as *hardware-enhanced* or *hardware-enabled security*) in order to enable *trustworthy computing. HW-assisted security* consists of technologies that provide security of higher layers of computer systems as firmware or software by using hardware components. It can enable security in BIOS, operating systems, hypervisors, or any other user-level application. It differs from hardware security, where solutions are restricted to protect physical devices or the hardware layer. Several architectural implementations of HW-assisted security solutions have been released by major vendors, like AMD, ARM, Intel, for a variety of scenarios, such as technologies for accelerated security-related processing, secure random number generation, memory bounds protection, isolated execution, video protection, or trusted computing. An overview of most relevant and promising HW-assisted security solutions are presented in [1].

According to [1], HW-assisted security solutions can be grouped into *Performance Boost* and *Security Enhancement* technologies as depicted in Figure 1.1. In the former group, the solutions deploy hardware to improve the performance of security processing or to improve security as a side effect, whose can be categorized as follows:

i) **Malware Detection** includes techniques for performance enhancement of anomaly detection processing;

ii) **Virtual Machine (VM) Isolation** increases the performance in virtualized systems protecting them against attacks as *hyperjacking* [5], *side-channel* [6] or *DMA* attacks [7];

iii) **Cryptographic Acceleration** embraces mechanisms to speed up performance of cryptographic processing;

In the latter group, the technologies use hardware to support security, which is categorized as:

i) **Pointers Violation Prevention** makes the system robust against *Code-Reuse Attacks* (CRA) such as *ROP* [8] or *JOP* [9], *control flow hijack* or *buffer overflow attacks* [10];

ii) **Random Number Generation** prevents attacks to the cryptographic schemes [11] that try to predict the randomly pattern of generated numbers used as digital signatures;

iii) **Trusted Computing** ensures protection of *data-in-use* using a physically separated hardware from CPU as *Trusted Platform Module* (TPM) or extra hardware integrated in the CPU like *Trusted Execution Environment* (TEE). It protects system against a subset of physical attacks like *bus sniffing*, or software ones like *code-injection* [12] or run-time attacks such as *Iago* [13];

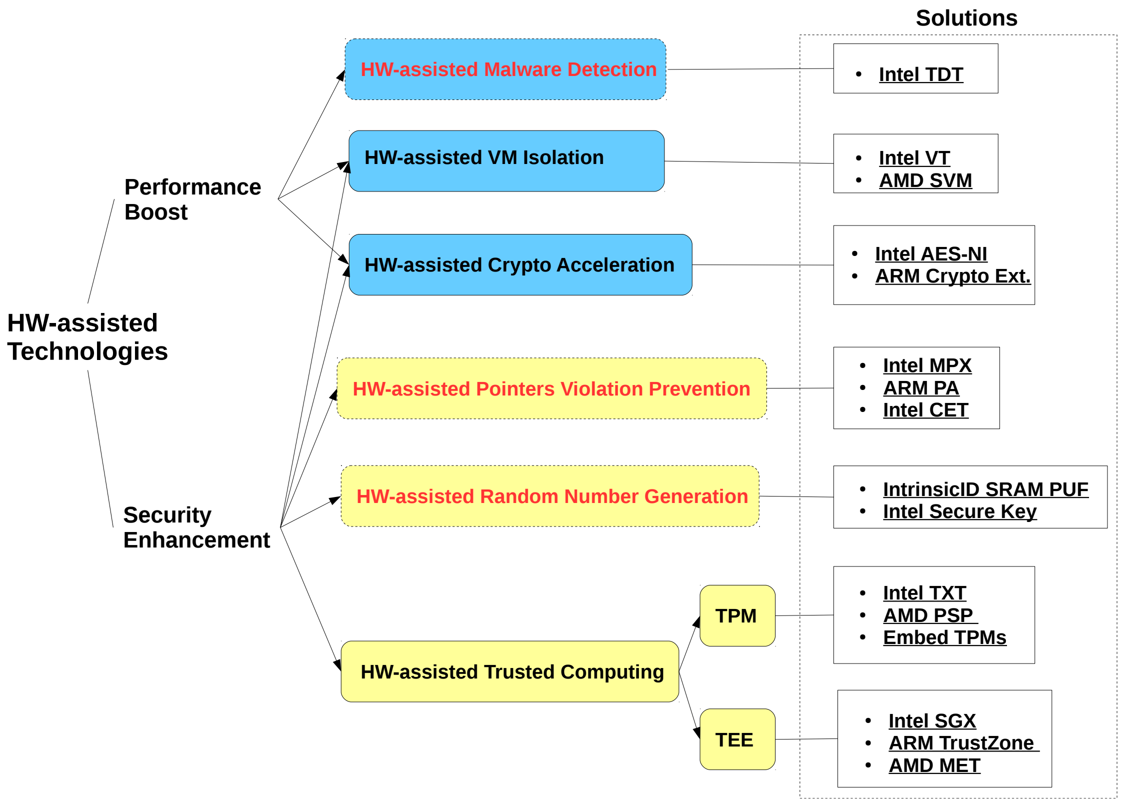


Figure 1.1: Category of HW-assisted security technologies [1]. This thesis contributes with optimization studies of solutions used in technologies classified in red-colored groups.

Cryptography relies on a secret key from storage system to protect the transmitted information. Conventionally, the key is stored in non-volatile memory (NVM) which is exposed by various security vulnerabilities of hardware primitives. There are demonstrations that an attacker with physical access to the storage can set up numerous attacks on it as side-channel attacks such as power measurement, semi-invasive attacks such as fault injection by over-clocking and invasive attacks such as decapsulation analysis to obtain information into storage [14, 15].

PUFs have emerged as appealing approach to protect secret keys and create device unique fingerprint or cryptographic key for confidentially, integrity and authenticity [15–17]. The main property of PUF is unclonability which relies on physically disorder system and impedes even the manufacturer to duplicate or clone the internal disorder circuits. It is possible due to the complex manufacturing fabrication of Integrated Circuits (IC). In theory, PUFs are secure against invasive attacks and distinct from each other.

Once authorized users have access to confidential data, there is no privacy guarantees that data cannot be leaked in subsequent computations. For example, Android users decide to grant personal information access (e.g. location, contacts, photos) to trusted app, but they have no assurance that the permissions will not be used to leak their sensitive information. To deal with this problem, there are some techniques developed to provide secure information flow, through either static analysis, where the source code or binary code is examined without executing it, or dynamic analysis which collects flow information during its execution.

Dynamic Information Flow Tracking (DIFT) have been established as promising platform to combat a wide range of security attacks [18]. The idea of DIFT is to tag (taint) an initial untrusted data and track its propagation during runtime. Data derived from untrusted data are also tagged and if any tagged data is used in unsafe operations, then an alarm is raised to block its execution. Initially, DIFT has been focused on explicit flow propagation which is related to data dependencies expressed by assignments. However, it has been demonstrated that an adversary can insert implicit flows based on control dependencies to execute evasion attacks [19].

Summing up, PUF and DIFT are effective mechanisms of HW-assisted security that improve security in ubiquitous environments which encompass constrained devices in terms of privacy and resources. As categorized in Figure 1.1, PUF belongs to the group of **HW-assisted Random Number Generation** and DIFT can be applied as technology for **HW-assisted Malware Detection** and **HW-assisted Pointers Violation Prevention**. Although they have been improved by extensively researches, there are still untreated security flaws and unsolved problems which demand new approaches and techniques to achieve the desired PUF and DIFT systems.

1.1 Contribution

This thesis proposes novel techniques related to PUF and DIFT fields in order to harden them as HW-assisted security technologies. As consequence of those research investigations, this work explore *Weightless Neural Network (WNN)* model as feasible HW-assisted security solution and use *Multi-Index Hashing* (MIH) data structure for similarity search problem on ubiquitous environment. The main contributions are listed below:

- Various novel PUF designs based on WNN are proposed with the aim of increasing their resistance against machine learning attacks. It is demonstrated that the architecture of neural networks can also be used to achieve security with low resource overhead.

- Robust and reliable PUF designs are proposed by extending the WNN PUFs. An initial entropy source is employed to create a near-ideal PUF in terms of reliability.

- Online techniques are developed to optimize a high-volume PUF testing solution based on MIH. The methods are proposed to reduce the storage space without negatively impact the performance.

- A new co-processor design to accelerate MIH operations is elaborated. It is evaluated in the similarity search problem on in-situ environment with low-cost FPGA.

- A novel DIFT technique is proposed to support implicit flows propagation, including scenarios with multiple nested branches. A formal verification is provided to prove its hardware design correctness. Moreover, it is practical to extend any DIFT mechanism that only supports explicit flow propagation.

- A restricted rule of implicit taint propagation is analyzed in order to increase the precision of DIFT mechanism. It reduces the number of data that are erroneously tagged during implicit propagation.

- New WiSARD (a simple WNN model) models based on *Approximate Membership Query* (AMQ) data structure are proposed. These models reduce the memory resources while improving the generalization capabilities of the WiSARD. Furthermore, AMQ models are robust as even in cases with high false positive cases they achieve good accuracy results. That fact indicates them as interesting solutions to provide high reliability in case of transient faults occurrences.

1.2 Thesis outline

The remainder of this thesis is organized as follows: Chapter 2 introduces the relevant concepts related to PUF, DIFT and WNN that are required to understand the other Chapters. Then, Chapter 3 presents many new PUF designs utilizing WNN which are resistant against machine learning attacks and proposes an extension for them to offer 100% of reliability by using a reliable entropy source. Chapter 4 defines techniques to optimize a design-for-test solution for testing PUF using MIH. Chapter 5 proposes a novel low resources DIFT technique to extend explicit DIFT with implicit flow propagation and describes a new implicit taint propagation rule to prevent data from being incorrectly tagged. Chapter 6 analyses new WiSARD models based on probabilistic data structures that perform membership query operations. Finally, Chapter 7 concludes this thesis and resumes the ideas for future works. Appendix A identifies the data-intensive MIH operations that can be efficiently implemented on hardware for future PUF testing implementation and evaluates the low-cost co-processor implementation for similarity search on in-situ environment. Appendix B presents complementary experiments for AMQ WiSARD models using Cuckoo filter and Quotient filter. Appendix C shows the list of publications submitted and accepted during the development of this thesis.

Chapter 2

Background

This chapter provides the relevant background information for better understanding of the addressed subjects in this thesis. The main topics are PUF, Dynamic Information Flow Tracking and Weightless Neural Network, while the remaining subjects are related to the concepts applied in the proposed solutions discussed later in this work.

2.1 PUF

Physically Unclonable Functions (PUFs) are circuits introduced by Pappu *et al.* as one-way functions which map inputs (challenges) to unique outputs (responses) [16]. PUF relies on manufacturing process variations where not even the manufacturer is able to clone or duplicate the physical components from one chip to another. It is possible due to the complexity of the current Integrated Circuits (IC) fabrication. Hence, in theory, PUFs are always distinct and unique between themselves and cannot be reconstructed by invasive attack. Such peculiar unclonability characteristic points out PUFs as a promising technology to protect secret keys.

2.1.1 Classification of PUFs

PUF can be classified into Weak PUF and Strong PUF according to the supported number of *challenge-response pairs* (CRPs). That classification also establishes the applications that each type of PUF is properly used.

2.1.1.1 Weak PUF

PUFs are classified into Weak PUF when the number of CRPs is limited, in some cases only a single challenge. That limitation makes Weak PUF unfeasible for authentication systems, as an adversary can easily collect all its CRPs to replicate it. On the other hand, Weak PUFs are better at generating and storing secret keys than non-volatile memory (NVM), since they offer harder accessibility from any invasive attacks to leakage internal information out. The generated secret keys are not secure against the side channel attacks. Therefore, the use of Weak PUF needs external countermeasures to hide it from applications that use its secret keys.

Weak PUF implementations based on SRAM cells have been studied extensively in previous researches [20, 21]. Each SRAM cell is an embedded memory consisting of six transistors as shown in Figure 2.1: two cross coupled inverters (load transistors T1, T2, T3, T4) and two access transistors (T5, T6) connected to the bit-lines (BLC and BL) and word-line signal (WL). When a SRAM cell is started-up, its state will transition to hold either 0 or 1 depending on noise and mismatch in the intrinsic process variations that are not controllable by the manufacturing process. By obtaining the values from the random stable states (0 or 1) of a SRAM array, it is possible to compose a physical fingerprint which in turn is used to create keys and identifiers. On the enrollment phase, the generated key is recorded as the correct key so that it will be matched to the next obtained keys whenever SRAM PUF is powered-up. Due to the intrinsic process variations, the power-up operations are affected by noise leading to the generation of incorrect keys and, consequently, the unreliability of the circuit. Some solutions to improve the reliability of Weak PUFs are discussed in Section 2.1.3.

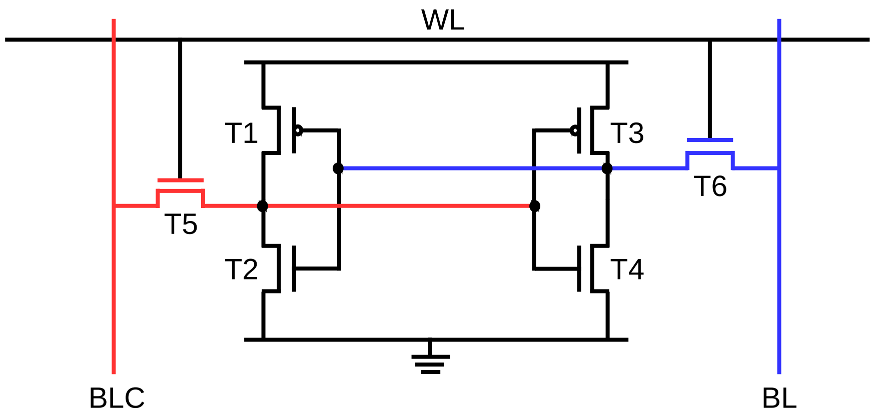


Figure 2.1: Example of six transistors at SRAM cell.

2.1.1.2 Strong PUF

Unlike Weak PUF, Strong PUFs support an exponential number of CRPs and implement a complex mapping between challenges and responses. Even an adversary has control over PUF, it is practically impossible to store all CRPs and predict the responses from the PUF. That complexity makes Strong PUF resistant to model building attacks and propitious for authentication applications using a Challenge

Response protocol. In the authentication, a set of CRPs from each PUF is previously stored into a secure database. When an authenticity is requested, a set of random CRPs from the database is applied to the PUF and the generated responses are compared to the stored responses in order to verify the result of the authentication. Once CRPs are chosen from database, they will never be reused for preventing man-in-the-middle attacks.

Unfortunately, one of the earliest Strong PUF implementations, termed Arbiter PUF [22], did not expose the prominent properties and could be easily cloned [23]. Proposed alterations to increase resistance, like using XOR operations, still failed to offer the promised unclonability [23]. Different from digital PUFs, analog circuits were proposed to harness non-linear behavior of CMOS transistors under certain operating conditions as a solution to increase the attack resistance of Strong PUFs. Current-based [24, 25] and voltage-based techniques [26] have been shown to be successful against Support Vector Machine (SVM) learning algorithm, which was able to break the previous digital PUF implementations. However, Vijayakumar *et al.* reported a new class of machine learning algorithms based on ensemble meta-algorithms that could effectively model even the analog PUFs with great accuracy [27]. Further, side-channel and fault-based attacks have also been explored to raise the modeling accuracy to break PUFs [28–30].

Other studies have been focused in the application of Weak PUF to construct Strong PUFs. Holcomb and Fu proposed a Strong PUF using SRAM cells organized in a column of a memory block which are pre-loaded with values based on an input challenge. To produce the final response, the PUF output is generated by reading multiple cells in a column at once to create a contention at the sense amplifier [31]. Bhargava *et al.* created a Strong PUF where the response is the cipher text generated by the AES block and the challenges are the plain text input which are extracted as a stable secret key from Weak PUFs [32]. Chapter 3 presents a reliable Strong PUF implementation based on neural network by utilizing similar concept of generating stable Weak PUF bits.

2.1.2 Ideal Properties of PUFs

The PUF implementations are built under circuits that harness the manufacturing variations and they differ by their physical implementations associated with the number of CRPs. To guarantee the quality of a PUF chip, it has to exhibit three ideal properties: high *uniqueness*, high *reliability* and high *security*.

2.1.2.1 Security

Security is the most important property of PUF directly related to unclonability and unpredictability requirements. It ensures the PUF are impossible to be physically cloned and built by software models that mimic the circuit. For Weak PUFs, it is implies to prevent the keys generated from PUF are discovered by external systems. In case of Strong PUF, the security depends on the complexity of challenge-response mapping. PUFs with simple mapping are easily predicted by software executing machine learning algorithms that creates a PUF model identical to the original circuit.

2.1.2.2 Uniqueness

As PUFs produce unique CRPs, the uniqueness determines how different the responses across distinct chips are. If PUFs generate similar responses, their challenge-response mappings are not ideal, straightly impacting the desired unclonability constraint. Therefore, high uniqueness is important to complement the security by defining the distinguishability of challenge-response mappings and discarding similar chips.

2.1.2.3 Reliability

PUFs might produce wrong responses under noise and certain environmental conditions such as supply voltage variations and temperature variations. Reliability is the property that indicates if the responses are stables along any such conditions. In Weak PUFs, reliability is critical since they are used to create secret keys or unique identifiers. To improve reliability, error-correction based solutions are applied to circuit by increasing the cost of design as cited in Section 2.1.3. The reliability issues in Strong PUFs can be mitigated by the fact that they have an exponential number of CRPs available, where a threshold of acceptable responses can be set for practical use in authentication. Increasing the threshold level also increases the number of CRPs needed to authenticate a device in real-world scenarios, resulting in decreasing in PUF reliability and raising in resource costs [33].

2.1.3 Weak PUF Realibility

As presented in Section 2.1.1.1, reliability is a crucial property for Weak PUFs enabling them to generate stable secret keys. To ensure Weak PUF reliability, techniques involving *error correcting-codes* (ECC) and *fuzzy extractors* have been extensively explored [34–38]. Nevertheless, such approaches use a large number of initial Weak PUF bits to derive the final stable key and this number increases with the inherent error rate of the Weak PUF [34]. Consequently, the implementation of ECC in hardware and the required number of Weak PUFs bits result in a large resource costs.

Other alternatives to costly ECC have been proposed, such as a simple circuit-based error correction using Temporal Majority Voting (TMV) [39, 40] and TMV improved by Vijayakumar *et al.* using Up-Down counters to correct twice the error rate, that is the correction rate comparable to other TMV implementation [41].

Modifications to SRAM Weak PUF circuit design have been explored to mitigate the error correction required and increase Weak PUF reliability [42–44]. New devices like Magnetic Tunnel Junction (MTJ) have also been studied to build reliable Weak PUFs [45]. Accelerated aging mechanisms have also been utilized to improve the reliability of Weak PUF [40, 46].

The wealth of techniques available to improve the reliability of Weak PUFs allows to create an efficient implementation that yields the desired number of stable bits for later to be used in neural networks to realize robust Strong PUFs as proposed in Section 3.3.

2.2 WiSARD

Weightless Neural Networks (WNNs) [47] are abstract neuron models which represent a neuron as Random Access Memory (RAM) node. These models offer an attractive practical solution to pattern recognition and artificial consciousness applications, due to their binary representation which able to implement such networks using existing memory resources in devices.

WiSARD (Wilkie, Stoneham and Aleksander's Recognition Device) is a multi-discriminator WNN model proposed in the early 80's [48] and inspired by the n-tuple classifier [49]. It is the pioneering WNN distributed commercially which provides simple and efficient implementation enabling to deploy learning capabilities into real-time and embedded systems.

Each class is represented by a structure called *Discriminator*, which comprises of a set of RAMs, composed of one-bit words, to store the relevant knowledge during the *training* phase which will be used during the *classification* phase. Before sending the input data to the discriminators, they need to be converted in a binary format using a transformation which depends on the data type. A binary input of $N \cdot M$ bits is split in N tuples of M bits. Each tuple n, $n = 1, \ldots, N$, is a memory address to an entry of the n-th RAM. Each Discriminator consists of multiple RAM blocks ($= N$), each containing 2^M locations. A pseudo-random mapping connects the tuples to the binary input and each Discriminator has its own pseudo-random mapping. A WiSARD system can have any number of such Discriminators and, hence, any number of desired classes.

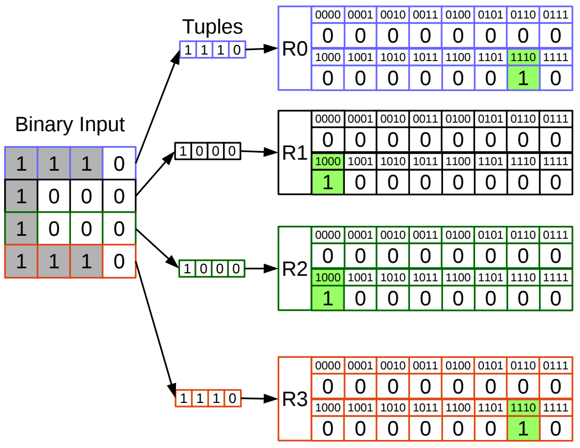


Figure 2.2: Example of training in WiSARD.

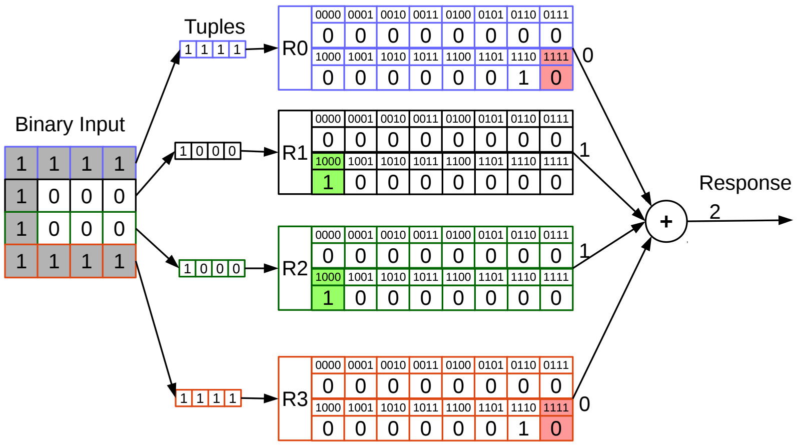


Figure 2.3: Example of testing operation in one WiSARD discriminator.

During the training phase, all RAMs of the Discriminators are initialized to zero (0). The training input is sent to the related Discriminator, where the accessed RAM positions are set to one (1) as illustrated in Figure 2.2. During classification, an input is sent to all Discriminators generating responses per discriminators by summing all accessed RAM values as shown in Figure 2.3. The Discriminator with the highest response is selected as representative class of the input as depicted in Figure 2.4. In both phases, the pseudo-random mapping from input to the tuples is the same for each discriminator.

The structure of WiSARD can be readily implemented in hardware using standard SRAM memory and address decoding to provide high generalization capabilities and real-time performance.

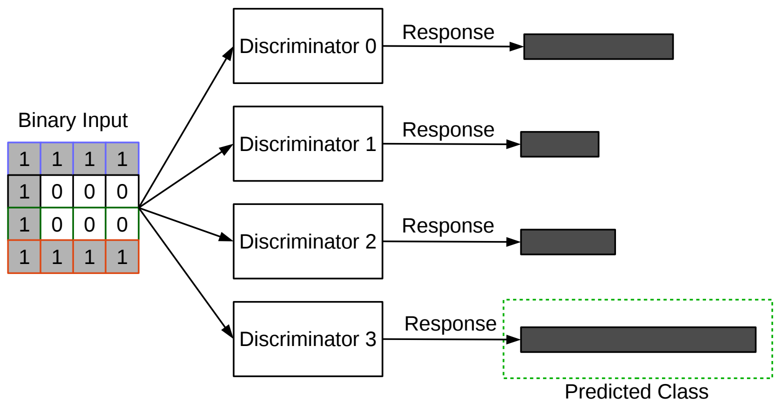


Figure 2.4: Example of testing operation to WiSARD select predicted class.

2.3 Dynamic Information Flow Tracking

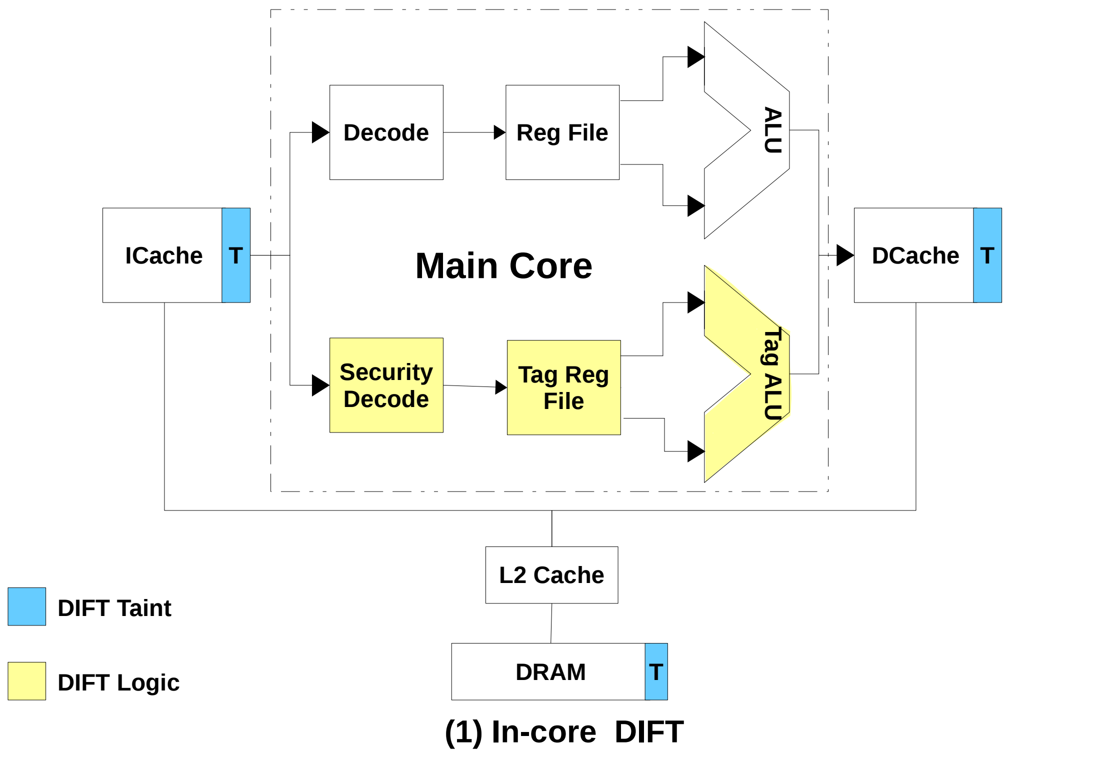
Dynamic Information Flow Tracking (DIFT), also known as *Dynamic Taint Analysis* (DTA), is a security enhancement technique applied to prevent sensitive information leakage and to protect binary codes against malicious attacks such as buffer overflows [50, 51] and SQL Injection [52, 53] at runtime. DIFT associates a tag termed *taint* to each memory word (or byte) and register, allowing to track information flow through taint propagation and taint checks using certain rules. Generally, the initial step is to tag the data coming from untrusted sources. During instruction execution, the taint will propagate from source operands to the destination operand according to the data dependency characterizing an *explicit information flow*. Checking the tainted data ensures safe execution and when the check fails a security alarm is raised to invoke the taint analysis. Depending on the type of analysis, program validation can be performed either directly by the DIFT system, or can be inferred from its behavior. In this way, security attacks are detected and sensitive information is kept safe by identifying unsafe activities and stopping the process execution.

2.3.1 Under-tainting and over-tainting problems

The *under-tainting* problem occurs when a data is not marked as tainted in cases that should be marked. For example, *implicit flow propagation* can be used to skip the taint propagation in control dependencies. Implicit DIFT mechanisms avoid this problem by tracking correctly such implicit flows.

In contrast to *under-tainting*, the *over-tainting* problem is the case that data is erroneously marked as tainted. Various implicit flow tracking approaches have limitations regarding nested branches and loops which intensify the taint propagation originated from control-dependency flow.

2.3.2 DIFT Designs



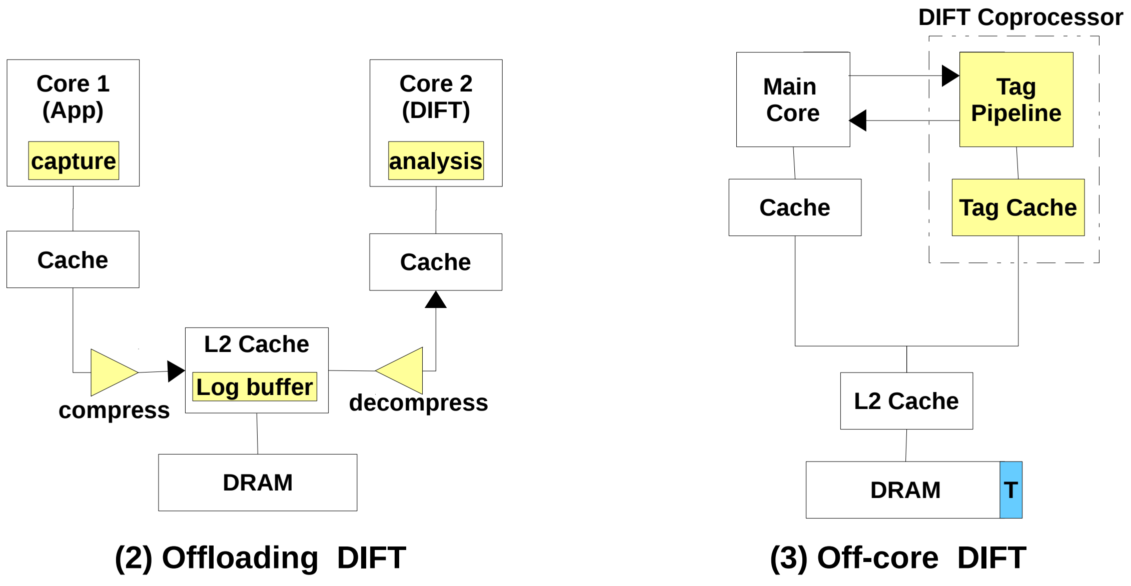


Figure 2.5: The three DIFT designs.

DIFT has been implemented in software [54–63] by using a binary instrumentation technique, that intercepts the program execution to insert extra instructions for tracking the tainted data at runtime, and hardware [51, 52, 64–71] by building a specialized circuit in order to reduce the performance overhead.

Figure 2.5 presents three hardware designs for supporting DIFT:

1. **In-core design**: DIFT logic is integrated in a general-purpose core by either

extending register file and memory resources, or adding specialized register file and cache to store the tags for data, respectively in the register and memory. Taint propagation and checking are performed in the processor pipeline which is extended to run in parallel with normal instruction execution. The performance impact is minimal in terms of clock cycles, but the required changes to the processor core may have a considerable negative impact on design and verification time. Most of the DIFT implementations are based on this approach [51, 52, 64].

2. **Offloading design**: DIFT functionality is performed by one core in a multicore chip while the application runs on another core. Although taint propagation and taint check policies are not implemented directly in hardware, the cores must still be modified to implement the synchronization scheme via system call between the application core and the DIFT core. The application core creates a compressed trace of executed instructions to communicate through a shared cache with the DIFT core, that decompresses the received trace in hardware before each tag execution. This design has been presented in [65, 66].

3. **Off-core design**: DIFT is implemented as a co-processor which has separated resources to manage the tags without any changes in the main core design. Synchronization between the main core and DIFT is needed to keep the correct tracking execution. Recent works have implemented DIFT as a co-processor [67–70].

2.3.3 Related Works

Most previous works have focused on explicit information flow tracking [51, 52, 59, 61, 64, 67–69] supporting multiple taint propagation and taint check policies. Software implementations on x86 architecture as LIFT [59] and libdft [61] rely on the use of Pin binary instrumentation framework [72], which intercepts the program execution to insert extra instructions for tracking the tainted data at runtime. Hardware approaches in [51, 52, 64, 67–69] propose to build a specialized circuit in order to reduce the performance overhead generated by DIFT computation. Raksha [52], Flexitaint [64] and [51] integrate DIFT logic in a general-purpose core by extending memories resources and processor pipeline. While in [67–69] proposed to implement DIFT as a co-processor which decouples resources of the main core design to manage the tainted data. These previous approaches show successful ability to track explicit flow, meanwhile they are limited to deal with *under-tainting* problem caused by implicit flows.

Similar to explicit flow solutions, software and hardware approaches have been proposed to solve *under-tainting* by tracking implicit information flow with control dependency. The major software solutions are DYTAN [62], DTA++ [63] and TASEL [60]. DYTAN taints all data belong to branch scope when control dependency is tainted. However, this approach causes *over-tainting* problem, where about $1,000$ times more data are tainted compared to explicit flow propagation as reported by [60]. DTA++ examines certain types of conditional branches to track implicit flow, that are within a specific code patterns based on the observation that *under-tainting* occurs at just few locations. TASEL elaborates a selective strategy to taint control-dependent data when used with tainted data inside the tainted branch. These solutions show effectiveness to track implicit flow in limited cases, because they do not consider tracking in multiply-nested branches scenario.

Recent hardware-based implicit flow tracking is presented by [70]. It designs an efficient implicit flow tracking unit (IFTU) built as a co-processor on a FPGA board, which keeps tracking implicit flows through the management of program counter tag register and stack. Program counter tag stack was proposed to correct tracking scheme in cases of multiply-nested branches. The approach discussed in Chapter 5 is inspired by IFTU implementation, where specialized taint registers are proposed to deal with implicit propagation in multiply-nested branches instead of using a program counter tag stack. Replacing the stack, the required resources are reduced for hardware implementation. Since DYTAN's taint propagation leads to *over-tainting* problem, a new taint propagation rule is suggested when a conditional branch is tainted to mitigate both *under-tainting* and *over-tainting* problems.

2.4 Error Correction Codes

Error correction codes have been applied in coding theory to correct corrupted data from channel noise, producing well-known results reported in [73]. Before transmitting a message $u = u_0, ..., u_{k-1}$ with k symbols through the channel, it is encoded into a codeword $v = v_0, ..., v_{n-1}$ of length n where $n \geq k$. The received codeword $r = r_0, ..., r_{n-1}$ might contain an error requiring a decoder to infer which message u was sent. A linear code is defined as code C over $\mathbb{F}_q$ such as all linear combinations of codeword resulting on codewords. Encoding a message u with linear code may be easily performed by multiplication of generator matrix G yielding codeword $v = u \cdot G$.

The most basic linear code is the repetition code. For the binary case, the codeword is formed by the redundant concatenation of each bit n times, where n is odd. The decoder is simply implemented by a majority vote algorithm and it can correct up to $(n-1)/2$ errors.

The Reed-Muller code is also a common linear code with a simple decoder construction using majority logic or the Hadamard transform. Despite the ease to implement, majority logic is not very efficient according to [74] and, therefore, the Hadamard Transform is usually used for decoding as defined in [73]. Reed-Muller code is described as $RM(r,m)$ code with order r and length 2^m. This work covers the first order $RM(1,m)$ code, where $n = 2^m$ is the length, $k = m+1$ is the dimension and $d = 2^{m-1}$ the minimum distance, equivalent to $[2^m, m+1, 2^{m-1}]$-code. The generator matrix of $RM(1,m)$ code is a $k \times n$ matrix. First row is vector $(\mathbf{1})$ filled with $1's$. The remainder m rows are vectors $(a_0), ..., (a_{m-1})$ with length n defined as $(a_i) = 11...100...011...1...00 = 1^{2^{m-i-1}}0^{2^{m-i-1}}1^{2^{m-i-1}}...0^{2^{m-i-1}}$, in the other words, (a_i) is formed by concatenation of consecutive words with $1's$ followed by words with $0's$ until to complete size n, both with length 2^{m-i-1}.

The decoder using Hadamard transform receives the codeword r and executes the following steps:

1. Generate R from r replacing 0 to 1 and 1 to -1.

2. Generate Hadamard transform
   $T = RH_{2^m} = (t_0, t_1, ..., t_{2^m-1})$.

   Where, $H_{2^m} = \begin{pmatrix} H_{2^{m-1}} & H_{2^{m-1}} \\ H_{2^{m-1}} & -H_{2^{m-1}} \end{pmatrix}$, $H_1 = (1)$.

3. Select t_i from T with largest magnitude.

4. Let index $i = (i_{m-1}, ..., i_0)_2$ in binary representation.
   corrected $\hat{r} = \sum_{j=1}^{m} i_{m-j} a_{j-1}$
   If $(t_i < 0)$ then $\hat{r} = \hat{r} + 1$

Note that $\hat{r}$ is the corrected codeword. The decoded code $u = x, i_{m-1}, ..., i_0$, where $x = \{0, 1\}$ according with the t_i signal. In Section 3.1.2.2, the x from decoded code u is calculated based on $i = (i_{m-1}, ..., i_0)_2$ to generate the final code of odd length. If $t_i \geq 0$ then $x = 0$ else $x = 1$.

Concatenated code was presented in [75]. The idea is to use simple concatenated codewords to create a long codeword. A message is encoded by an outer encoder, which in turn is split and each piece is encoded by an inner encoder. The final codeword is composed by concatenating all inner codewords. Thus, the decoder may be easily implemented by merging both inner and outer decoders. The short parts of a received codeword r are decoded by the inner decoders in parallel producing an outer codeword, that is decoded by outer decoder.

2.5 Multi-Index Hashing

Multi-index hashing (MIH) is a fast search technique to obtain the nearest neighbors in Hamming space [76]. MIH offers high throughput by storing a binary code into m hash tables indexed with its m disjoint sub-codes. During searching, the query is split into m sub-codes so that the neighbors candidates are parallelly sought over the hash tables. Then, all candidates are validated by looking the original query to remove any false neighbor, ensuring exact kNN search over sub-linear run-time.

This approach relies on the idea that two similar binary codes also contain similar sub-codes and, thus, by finding the candidates with similar sub-codes reduces the search space and saves memory space. MIH algorithm is formulated by the following premise. Let us suppose two binary codes h and g both with b bits are partitioned into m disjoint binary sub-codes with s bits length, where $s = \lfloor \frac{b}{m} \rfloor$ or $s = \lceil \frac{b}{m} \rceil$ and b divisible by m. When h and g differ by r bits or less, then at least one sub-code k must differ by $\lfloor \frac{r}{m} \rfloor$. That is, there must be a sub-code k, $1 \leq k \leq m$, where

$$\parallel h^{(k)} - g^{(k)} \parallel_H \leq \lfloor \frac{r}{m} \rfloor \tag{2.1}$$

when $\parallel h - g \parallel_H \leq r$ with $\parallel . \parallel_H$ denoting Hamming norm. The proof of the premise is derived from Pigeonhole Principle [76].

To exemplify the use of MIH, let us suppose the database stores 64-bit binary codes and, given a 64-bit query q, all nearest neighbors with until 16 Hamming distance from q are searched. Also, let us suppose that MIH has 4 hash tables where each one stores 64-bit binary codes indexed by their subsequent parts of $64/4 = 16$ bits. Each query is divided into 16-bit sub-strings and each sub-string is a key to access the list of 64-bit binary code which contain it. As generalized by Equation 2.1, if 64-bit binary code b and q differ by at most 16 bits, it means that at least one of the associated sub-strings differs at least $\frac{16}{4} = 4$ bits. According to this premise, a list of candidates can be quickly retrieved by executing search with radius $r = 4$ to find 4-neighbors over each sub-string in the corresponding hash table. Thus, the number of lookups is reduced from $\binom{64}{16} \approx 4,9 \times 10^{14}$ to $4 \times \binom{16}{4} = 7,280$. At the end, each candidate is verified as true 16-distant neighbor to ensure the exact neighbor match. This approach provides high order of magnitude speedup with the ability to search millions of 128-bit codes within a second using a search radius of 30 bits [76]. Moreover, the algorithm ensures to find exact neighbors without any approximation.

Chapter 3

Reliable Strong PUFs based on Weightless Neural Network

Electronic devices are increasingly used in applications like Internet of Things (IoTs) and potentially have more access to sensitive information while running in untrusted environments. Due to resource constraints, by designing hardware roots of trust (RoT) is an attractive alternative to integrate security into IoT devices such as *lightweight* authentication operations and secret key generation capabilities. PUFs are one class of lightweight roots of trust that require high uniqueness and reliability to provide robust security as introduced in Section 2.1.

Nevertheless, practical Strong PUF implementations may suffer from reliability issues [24, 26] and, furthermore, they are susceptible to be cloned with high accuracy by machine learning attacks [23, 27]. In authentication applications, unreliability can affect the number of CRPs required to properly distinguish a PUF-equipped integrated circuit (IC) from billions of other such devices [33]. Thus, there is still a need to design Strong PUFs to offer fully reliability and immunity against model building attacks using machine learning techniques.

Recently, neural networks have been broadly studied for their effectiveness at pattern recognition applications. WiSARD is a Weightless Neural Network (WNN) with efficient and simple implementation by using random-access memory (RAM), as introduced in Section 2.2. This Chapter explores adapting the simple WiSARD architecture to create Strong PUFs with high uniqueness and high level of resistance against machine learning attacks. In addition, the reliable Weak PUFs are combined to create reliable Strong PUFs. Reliability of a Weak PUF is critical as detailed in Section 2.1.3 and has received widespread attention [34–36, 38, 40, 41].

First, WiSARD WNN model is combined to the SRAM PUF properties to realize various Strong PUF designs. As introduced in Section 2.1.1.1, SRAM cells are the most promising choice for creating Weak PUFs due to their stability, yet random power-up values [20, 21]. Later, an initial entropy source consisting of a set of highly reliable Weak PUF is employed to load the bits into the contents of the WiSARD WNN RAMs in order to provide a robust Strong PUF architecture. Since the initial Weak PUF bits are made reliable, this reliability is extended to the final Strong PUF.

3.1 Strong PUFs based on Weightless Neural Network

Next Sections, all Strong PUF designs inspired by the WiSARD model are presented through architectures that produce a 1-bit output response, given an m-bit input challenge. The extended versions are examined with the objective of improving the resistance against machine learning attacks.

3.1.1 WiSARD PUF

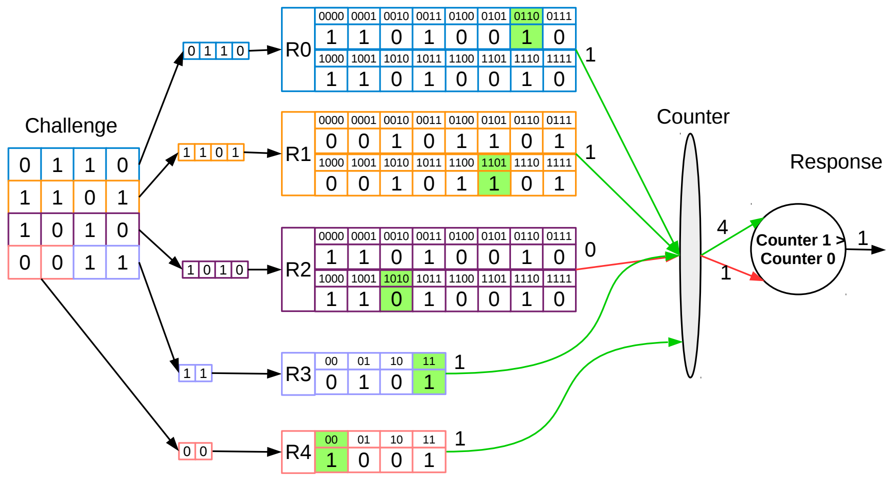


Figure 3.1: Example of WiSARD PUF architecture [2].

The first design is termed as *WiSARD PUF* and depicted in Figure 3.1. It is a single Discriminator containing multiple RAM blocks where the input is a challenge string. The challenge bits are sliced in a pseudo-randomic way into tuples. Each tuple indicates the size of the RAM block Ri and forms the address of a unique block. Thus, the 1-bit memory locations from the RAM blocks are accessed according to the input challenge and the accessed values are sent to a counter that accumulates the number of 1's. The final 1-bit response is generated through the majority voting over the accumulated values. For the proper use of the majority voting, it is required to have an odd number of RAM blocks. When the number of tuples is even, it can be accordingly adapted to create an odd number of RAM blocks by splitting one tuple in half and associating the extra tuple to a new RAM. This case is exemplified in Figure 3.1, where 16-bits input challenge is mapped to three 4-bit tuples and two 2-bit tuples to complete the total of 5 RAM blocks.

WiSARD PUF operates similarly to the WiSARD *classification* phase to produce the final response, which counts the accessed RAM bits instead of summing those bits to calculate the Discriminator response. Other different aspect from conventional WiSARD is the absence of *training* phase. The RAM blocks are initialized with random values when powered on in the same way as SRAM PUFs [20, 21], since they are formed of SRAM cells whose ensure such property as detailed in Section 2.1.1.1. Different instances of WiSARD PUF can hold their own pseudo-random mapping of challenge to tuples and their random RAM contents. Since WiSARD PUF contains multiple SRAM cells, its reliability is affected by the noises coming from the SRAMs cells along multiple power-ons [77]. Therefore, it is indispensable to ensure an acceptable Strong PUF reliability.

By designing the hardware, the unique pseudo-random mapping implementation for each PUF can be costly in terms of area. Strategies with multiple mapping can be used, but are considered beyond the scope of this work. To simplify, WiSARD PUFs with a fixed mapping decided by designer across all instances are analyzed, as depicted in Figure 3.2. The subsequent architectures that will be presented in the next Sections assume a fixed mapping of the input challenge to tuples across all PUF instances.

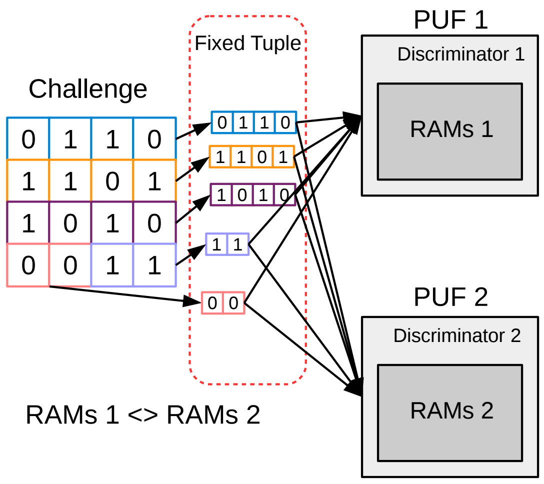
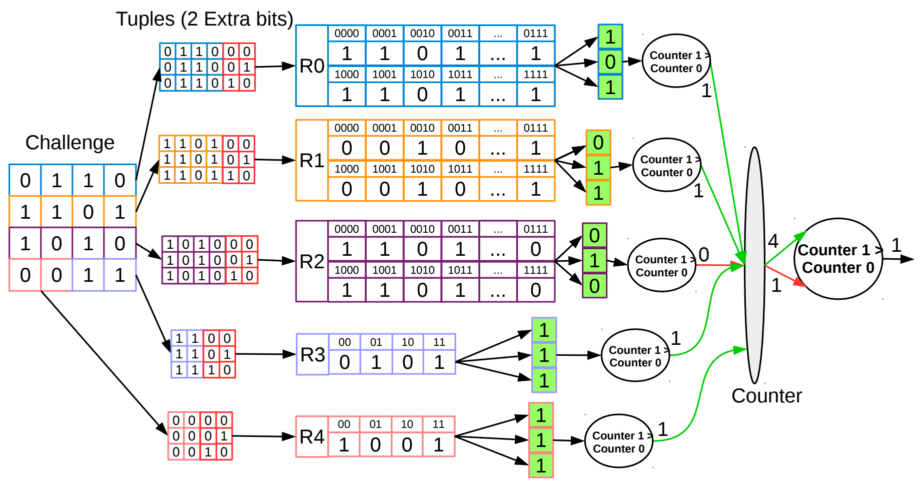


Figure 3.2: Example of WiSARD PUF with fixed tuples among PUFs [2].

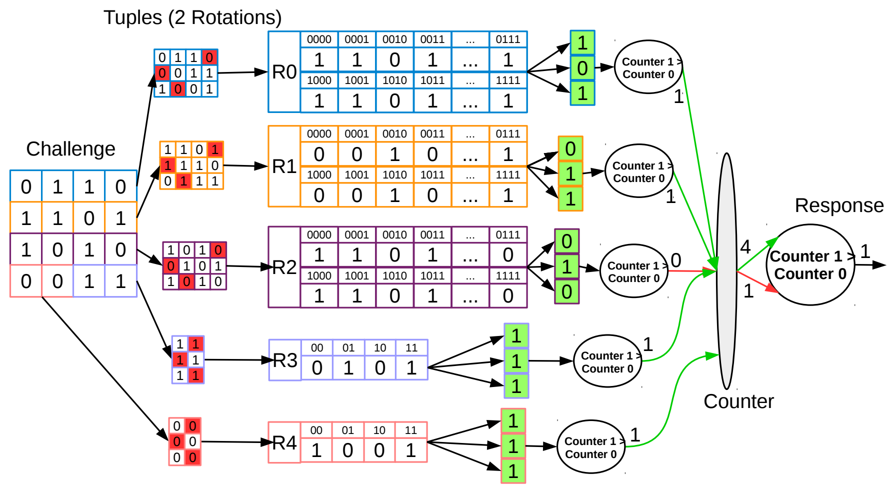
3.1.2 Extensions to WiSARD PUF architecture

Using a simple majority voting on the RAM block outputs to produce WiSARD PUF response is adequate to create a Strong PUF. Other possible extensions are explored with the aim of improving machine learning attack resistance in comparison with the original WiSARD PUF. In particular, the modifications are applied to either the tuple generation for addressing the RAM blocks or the output processing of the blocks to generate the final output response.

3.1.2.1 Fuzzy logic based address generation



(a) Example of WiSARD PUF architecture with extra bits [2].



(b) Example of WiSARD PUF architecture with tuple rotations (circular shifts) [2].

Figure 3.3: Example of WiSARD PUF architecture with extensions to address generation [2].

A popular solution to deal with noisy data is to harness fuzzy extractor, as evidenced by its advantageous use in deriving stable keys from biometric data [78, 79]. Fuzzy logic has been widely used to increase Weak PUF reliability in the presence of noise [34, 36–38]. Fuzzy extractor is usually performed in two phases: *enrollment* phase, which creates *helper data* from manipulating the input data in a trusted environment and *reconstruction* phase, where the received data is assumed to be noisy and the proper helper data is used to retrieve the error-free response originally used in the enrollment phase.

The concepts of fuzzy extractor were utilized to generate more data from RAM blocks than in a normal WiSARD PUF, akin to helper data generation, and reduce this extra data to obtain the final PUF response, analogous to reconstruction. To collect multiple outputs from each RAM block, two approaches were adopted to affect the challenge tuples – (a) add *extra bits* in random locations to each tuple; (b) perform *rotation* (or *circular shift*) operation on each tuple.

In extra bits approach, e extra bits are added to each tuple allowing to generate the total of 2^e combinations of the extra bits and, correspondingly, the same number of addresses and outputs from each RAM block. All combinations can be generated internally using a simple counter. For the rotation approach, a predefined number of *right circular shift* (or rotation) operations is performed to each *original* challenge tuple to generate multiple outputs from the corresponding RAM block using the new addresses.

Both methods increase the system entropy for the same input challenge to further help protect the PUF against machine learning attacks. Figure 3.3(a) illustrates a WiSARD PUF with 2 extra bits while Figure 3.3(b) illustrates an example using 2 tuple rotations. In those examples, a 16-bit input challenge is applied so that each RAM block generates 3 outputs which undergo majority voting to produce a single output per block, following to the second majority voting process to produce the final 1-bit response. The required odd number of inputs for the majority voting is guaranteed either by performing an even number of tuple rotations, or by adding $(2^e) - 1$ combinations of the extra bits, whereas the required odd number of RAM blocks is obtained in the same way as the basic WiSARD architecture.

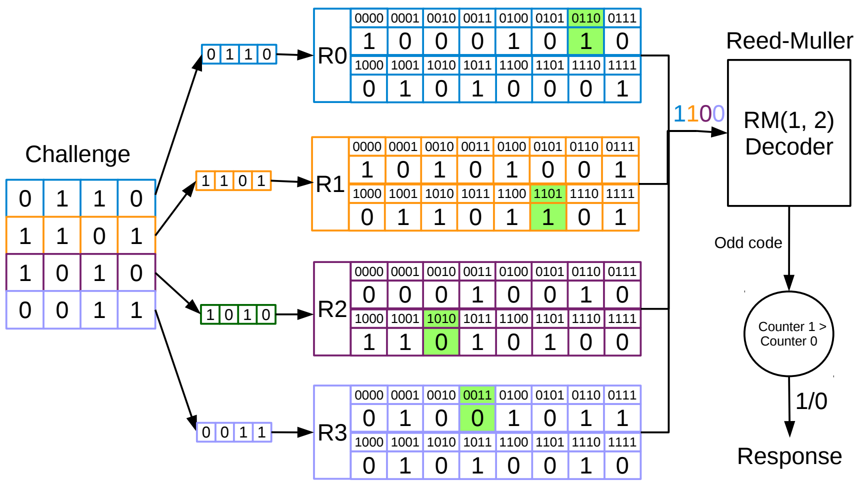


Figure 3.4: Example of RM-WiSARD PUF architecture [2].

3.1.2.2 Concatenated codes based response generation

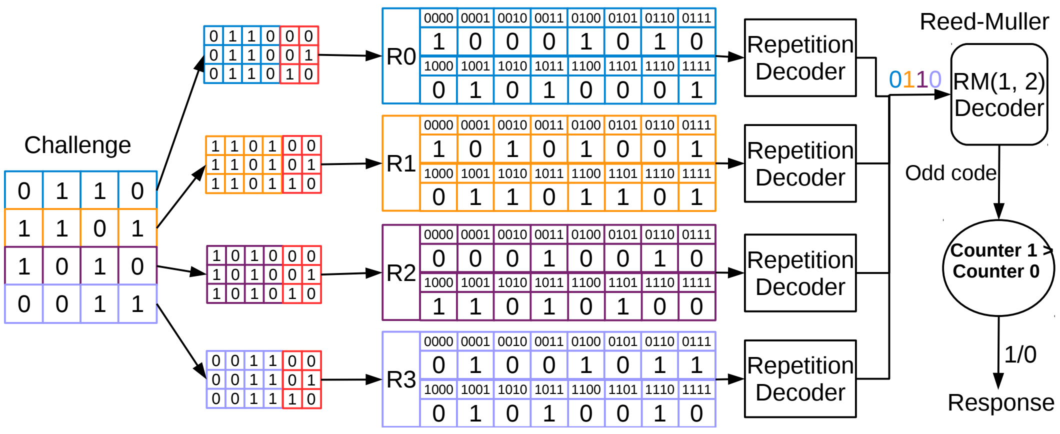
The majority voting utilized for response generation in the WiSARD PUF architecture is akin to using a *repetition* code, as shown in Figure 3.1. Other *error-correcting codes* (ECC) can also be implemented for the purpose of generating the final response from the RAM block outputs. In contrast to the linear relationship exhibited by majority voting, coding scheme can introduce a non-linear relationship between the RAM outputs and the response, so that it can greatly benefit the machine learning attack resistance of the final Strong PUF. Christoph Bösch illustrated the advantages of concatenated ECC to improve Weak PUF reliability and also provided details of hardware implementation using various ECC schemes [34]. One of the *concatenated ECC* schemes was employed to the basic WiSARD PUF architecture. A brief introduction about ECC theory is presented in Section 2.4.

A new design termed as *RM-WiSARD* architecture applies *Reed-Muller* (RM) code-based decoder to the RAM block outputs, as exemplified in Figure 3.4. The repetition code is performed on the output bits from RM decoder to produce the final response. $\mathrm{RM}(1, m)$ decoder is adopted in PUF design that receives an input of length 2^m and generates an $(m+1)$-bit output. This modifies the architecture to contain 2^m RAM blocks able to produce the necessary RM inputs. The decoder implementation is based on Hadamard transform algorithm (detailed in Section 2.4) with a simple modification to extract a final code of odd length. Reed-Muller decoder can be efficiently implemented on hardware by scaling with the chosen value of m as explained by Christoph Bösch [34].

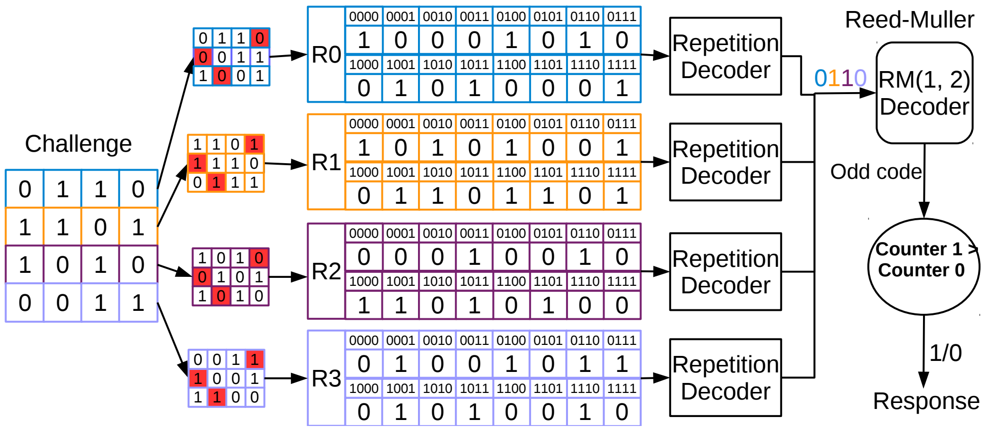
RM-WiSARD PUF can be extended through coupling extra bit or tuple rotation mechanisms. It is also modified to accommodate the final PUF response generation through concatenated ECC scheme. Figure 3.5(a) illustrates an example of the RM-WiSARD implementation with extra bits and Figure 3.5(b) shows a RM-WiSARD implementation with tuple rotations where, in both variations, the repetition code is applied to the outputs from each RAM block. Then, the results are sent to RM decoder to produce an output with odd number of bits. At the end, the repetition code is used again to obtain the final 1-bit response.

3.2 WNN PUF - Experimental Setup and Results

The proposed designs presented in Section 3.1 are evaluated to identify which one can provide the highest machine learning attack resistance. Additionally, the inter-class (uniqueness) and intra-class (reliability) Hamming distance metrics are analyzed to further compare and contrast the suitability of various designs as Strong PUFs.



(a) Example of RM-WiSARD PUF architecture with concatenated code (extra bits)[2].



(b) Example of RM-WiSARD PUF architecture with concatenated code (tuple rotations)[2].

Figure 3.5: Example of RM-WiSARD PUF architecture with concatenated code [2].

3.2.1 Experimental Setup

The experiments for all PUF designs are simulated in Python and the relevant SRAM circuit data is obtained from SPICE simulations using 45 nm transistor models [80]. Each PUF receives a 64-bit input challenge to produce a 1-bit response. The Discriminator SRAM cells corresponding to each PUF instance are initialized with the random power-on states such that the average number of 1's and 0's are equal. Table 3.1 contains the general details about the WNN designs. For designs with 9 RAM blocks, 7 are addressed by 8-bit tuples while the remaining two use 4-bit tuples. The extended address generation designs assume 2 extra bits and 2 rotations for the respective implementations. RM$(1,3)$ decoder is utilized for all RM-WiSARD PUF design variants. Except original WiSARD PUF, all designs assume a fixed input-to-tuple mapping across PUFs, as depicted in Figure 3.2.

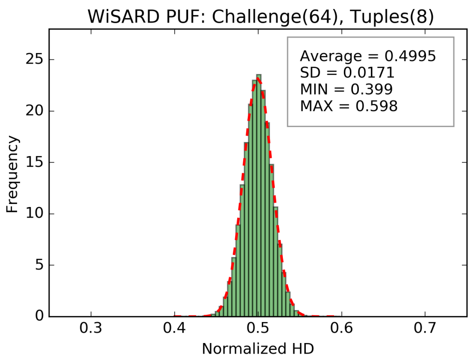
Table 3.1: WiSARD PUF design architectures for 64-bit challenges [2].

	#RAM Blocks	*#Addresses*	*#SRAMs*
WiSARD PUF	9	$7 \cdot 2^8 + 2 \cdot 2^4$	$1,824$
WiSARD PUF (fixed tuple)	9	$7 \cdot 2^8 + 2 \cdot 2^4$	$1,824$
RM-WiSARD PUF	8	$8 \cdot 2^8$	$2,048$
WiSARD PUF + 2 Extra bits	9	$7 \cdot 2^{10} + 2 \cdot 2^6$	$7,296$
WiSARD PUF + 2 rotations	9	$7 \cdot 2^8 + 2 \cdot 2^4$	$1,824$
RM-WiSARD PUF + 2 Extra bits	8	$8 \cdot 2^{10}$	$8,192$
RM-WiSARD PUF + 2 rotations	8	$8 \cdot 2^8$	$2,048$

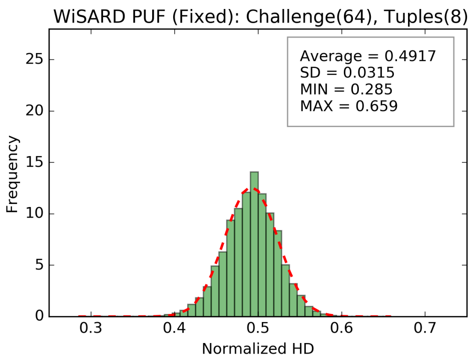
3.2.2 Uniqueness

Uniqueness is a property wherein indicates the level of differences among the responses produced from various PUF instances when they receive the same challenge. Inter-class Hamming distance (HD) is the metric used to determine uniqueness, which calculates the Hamming distances between the responses of each pair of PUFs for the same challenge. Then, the average of inter-class HD, d_{inter}, is calculated considering the HD results over many challenges and PUF instances according to the following formula:

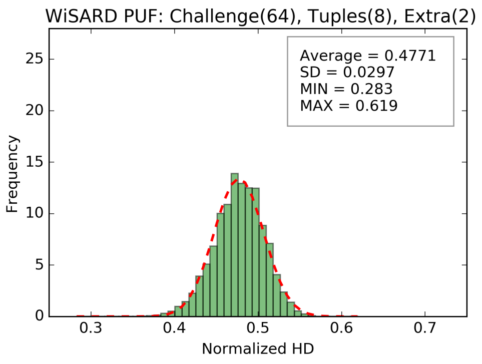
$$d_{inter} = \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{HD(r_i, r_j)}{n} \tag{3.1}$$



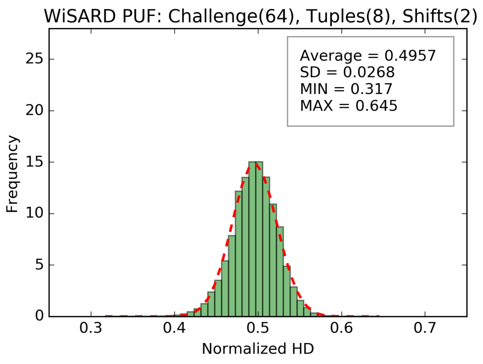
(a) WiSARD PUF.



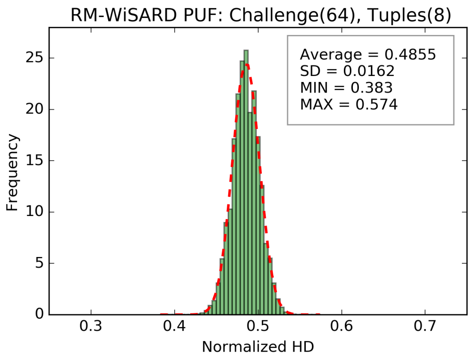
(b) WiSARD PUF with Fixed Tuples.



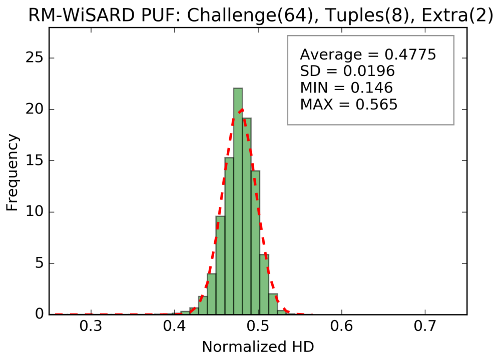
(c) WiSARD PUF with extra bits.



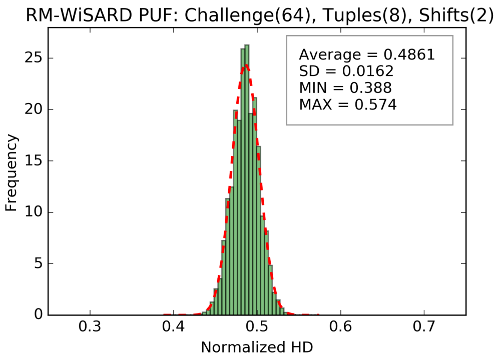
(d) WiSARD PUF with tuple rotation.



(e) RM-WiSARD PUF.



(f) RM-WiSARD PUF with extra bits.



(g) RM-WiSARD PUF with tuple rotation.

Figure 3.6: Inter-class Hamming Distance (uniqueness) distribution for WiSARD PUFs.

where k is amount of PUFs, n is the total of bit responses and $HD(r_i, r_j)$ is Hamming distance between responses of the PUF instances i and j to a particular challenge. A PUF is considered ideal when the normalized inter-class HD = 0.5, that is, on average 50% of the response bits are different.

For this experiment, ($k =$) 1000 PUF instances were evaluated for each design, ($n =$) 1000 challenges were applied to each PUF to produce the total of 1000 response bits. Figure 3.6 presents the uniqueness results for the several PUF designs. All normalized inter-class HD distributions behavior akin Normal distribution clustering the distances around the average with small standard deviation.

Overall, every designs achieved the mean of normalized inter-class HD close to the ideal 0.5 (almost 500 bits are different), while the original WiSARD PUF (random input-to-tuple mapping) resulted in the highest normalized HD. Between the extended address generation designs, the tuple rotation offered better results in comparison to extra bits approach. Additionally, the RM-WiSARD variants have smaller standard deviation, close to the original WiSARD PUF. While the fixed tuple mapping of challenges was adopted for the majority of the WiSARD PUF designs, an interesting point is explore random challenge mapping in hardware in order to obtain an efficient way to provide greater uniqueness to the model.

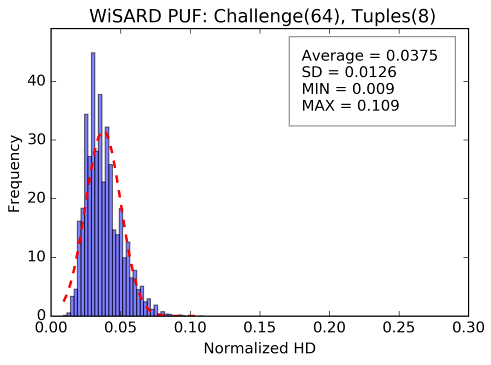
3.2.3 Reliability

Reliability is the property to ensure the generation of the correct responses given the same challenges in any condition variations, such as in the presence of noise. Intra-class Hamming distance is the metric used to determine reliability, that calculates the Hamming distance between the correct responses (r_i) extracted from a PUF under ideal conditions and the set (m) of noisy responses (r'_i) by introducing errors in RAM locations across multiple power-ons. Then, the average of intra-class HD, d_{intra}, is calculated over the HD results as:

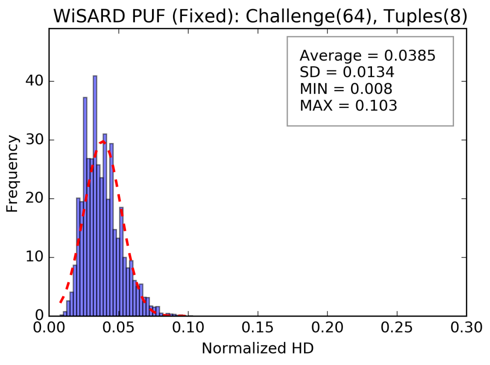
$$d_{intra} = \frac{1}{m} \sum_{t=1}^{m} \frac{HD(r_i, r'_{i,t})}{n} \tag{3.2}$$

where n is the number of CRPs collected from each PUF and $HD(r_i, r'_{i,t})$ is Hamming distance between responses of the ideal PUF instance i and its t-th noisy instance to a particular challenge. A PUF is considered ideal when the normalized intra-class HD = 0 for any challenge, representing 100 % reliability. In other words, on average 0% of the response bits are different or 100% of the response bits are equals.

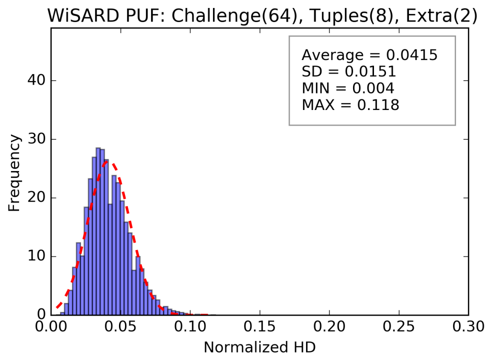
The noisy conditions are simulated by considering each SRAM cell has an embedded inherent error rate across multiple power-ons. Roel Maes introduced heterogeneous error modeling with cell-specific error probabilities to evaluate the reliability



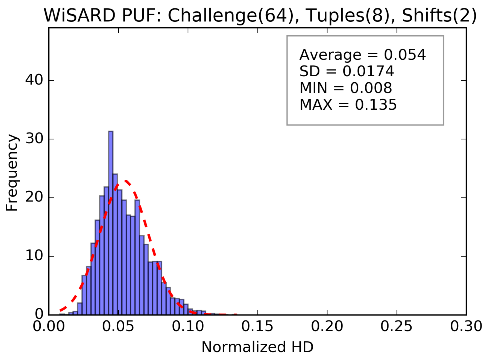
(a) WiSARD PUF.



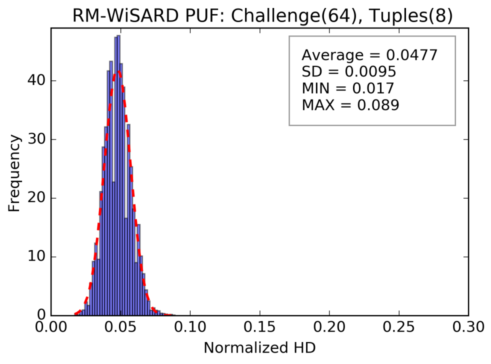
(b) WiSARD PUF with Fixed Tuples.



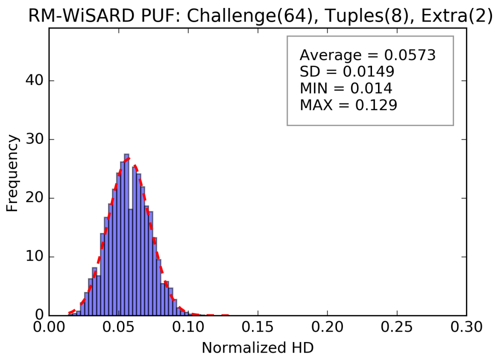
(c) WiSARD PUF with extra bits.



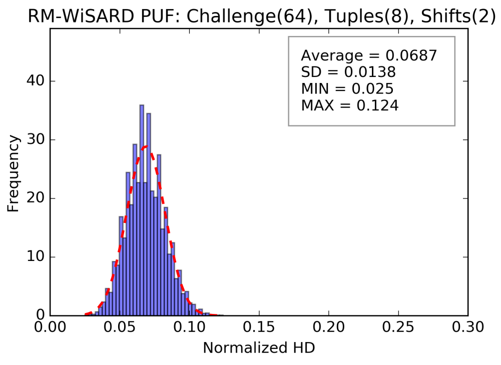
(d) WiSARD PUF with tuple rotation.



(e) RM-WiSARD PUF.



(f) RM-WiSARD PUF with extra bits.



(g) RM-WiSARD PUF with tuple rotation.

Figure 3.7: Intra-class Hamming Distance (reliability) distribution.

of PUFs with high accuracy [77]. The proposed 2-parameter error model associates an error probability $p_{e,i}$ at each SRAM cell and defines a random variable P_e to sample all values of $p_{e,i}$, according to the distribution:

$$cdf_{P_e}(x) = \lambda_1 \cdot \int_{-\infty}^{\Phi^{-1}(x)} \Phi(-u) \cdot (\varphi(\lambda_1 u + \lambda_2) + \varphi(\lambda_1 u - \lambda_2))du$$

where $\Phi(x)$ is *cdf* of Standard Normal Distribution, $\varphi(x)$ is *pdf* of Standard Normal Distribution and λ_1 and λ_2 are input parameters. $10,000$ SRAM cells were simulated in 45 nm CMOS technology [80] in the presence of thermal noise and the error rates for each instance were obtained across 1000 power-ons. The data was curve-fitted to obtain the input parameters, found to be $\lambda_1 = 0.2916$ and $\lambda_2 = 1.9062$. Thus, the error rates for an arbitrary number of SRAM cells of WNN PUFs are generated by applying the methodology proposed by Maes [77].

The WiSARD PUF designs are simulated to produce the responses across multiple power-ons applying the same challenges. For each design, 100 PUF instances were analyzed where each one received ($n =$) 1000 challenges and for each challenge ($m =$) 100 noisy responses were generated by simulating multiple power-ons for the SRAM cells. Figure 3.7 shows the reliability results for the various PUF designs. Original WiSARD PUF performs better than all the other variants. In contrast with the uniqueness results, the extra bits offered better reliability in comparison to tuple rotation approach. It is possible further improve reliability by utilizing different processing schemes for the RAM block outputs. Those schemes will be targeted in the future works.

3.2.4 Machine Learning Attack Resistance

Machine learning techniques like Logistic Regression (LR) and Support Vector Machine (SVM) have demonstrated the ability to clone previous digital Strong PUF designs [23]. Advanced machine learning (ML) techniques based on ensemble meta-algorithms, like Gradient Boosting, were successful committed to break even analog Strong PUFs by Vijayakumar *et al.* [27], that were initially resistant to SVM. Therefore, LR, SVM and Gradient Boosting (Grad Boost) are used to measure the attack resilience of the proposed PUF designs.

The machine learning algorithms were implemented in Python using the scikit-learn tools [81]. Gradient Boosting was configured with the number of estimators set at 128 and learning rate of 0.01. For LR, the inverse of the regularization strength was set to a value of 10^{-5}. SVM utilizes radial basis function (RBF) kernel machines to model non-linearly separable functions as linearly separable in higher dimensions. For each design, 100 PUF instances were analyzed and $150,000$ CRPs were collected from each PUF instance of which 100,000 CRPs were used for *training* to obtain the cloned PUF model and 50,000 CRPs were applied for *testing* the machine learning attack accuracy. A Strong PUF is considered ideal when machine learning techniques achieve low accuracy to learn its challenge-response mapping. A chip cloned with on average of 50% of accuracy by these techniques is acceptable since it is akin to random guessing.

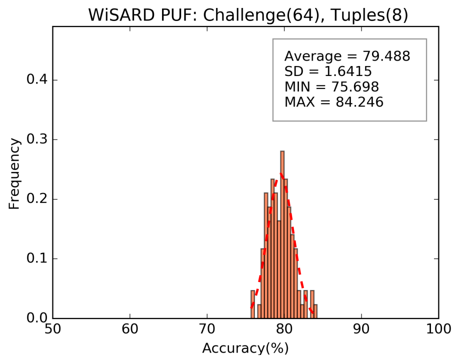
Table 3.2 tabulates the *mean* and *standard deviation* of accuracies obtained from discussed ML algorithms for each PUF architecture. Gradient Boosting achieved the highest learning accuracy to clone the models and all RM-WiSARD PUF variants offered high machine learning resistance (lower accuracies) across the various learning algorithms with 'RM-WiSARD PUF with Tuple rotation' providing the best results. Figure 3.8 presents the accuracy distribution of Gradient Boosting, as it had the highest learning accuracy results compared to SVM and LR. RM-WiSARD with tuple rotation design obtained the lowest accuracies with small standard deviation indicating it as the most resistance design against machine learning attack. Applying extra bits or tuple rotation modifications to the original WiSARD PUF worsened the modeling attack resistance for those PUFs. Thus, the output generation of RAM blocks has to be addressed in order to determine how improve attack resilience.

Table 3.2: Machine Learning results for WiSARD PUF variants [2].

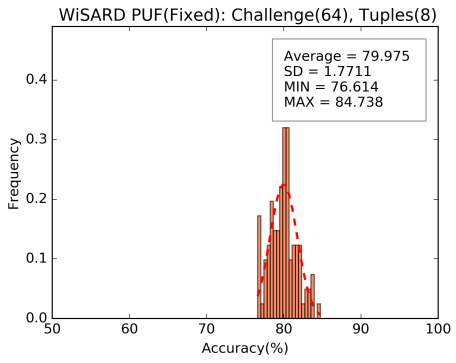
PUF Type	**Machine Learning Accuracy**					
	Grad Boost		**SVM**		**LR**	
	μ	σ	μ	σ	μ	σ
WiSARD PUF	0.79	0.016	0.690	0.020	0.637	0.246
WiSARD PUF (fixed tuple)	0.790	0.017	0.687	0.026	0.630	0.032
RM-WiSARD PUF	0.612	0.028	0.585	0.01	0.583	0.048
WiSARD PUF + Extra bits	0.815	0.018	0.722	0.028	0.652	0.033
WiSARD PUF + Tuple rotation	0.822	0.019	0.662	0.017	0.600	0.023
RM-WiSARD PUF + Extra bits	0.667	0.047	0.61	0.04	0.602	0.039
RM-WiSARD PUF + Tuple rotation	0.594	0.011	0.584	0.008	0.584	0.008

3.2.5 Hardware Analysis

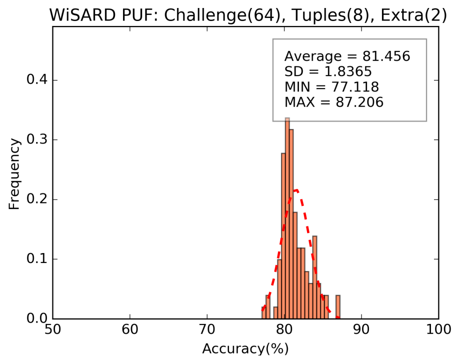
Since all proposed designs in Section 3.1 integrate a Weightless Neural Network hardware and extra resources for concatenated code and fuzzy logic versions, the



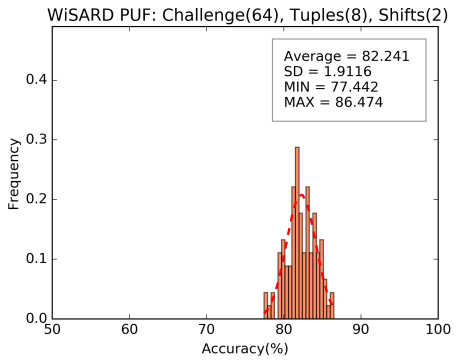
(a) WiSARD PUF.



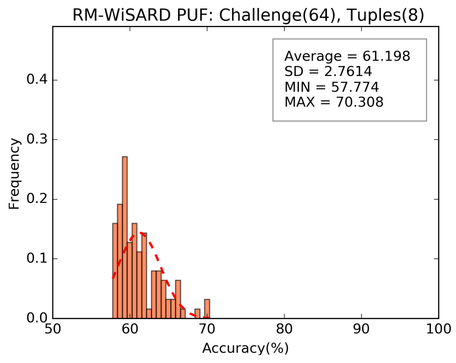
(b) WiSARD PUF with Fixed Tuples.



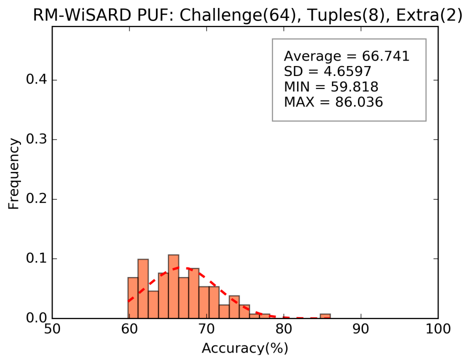
(c) WiSARD PUF with extra bits.



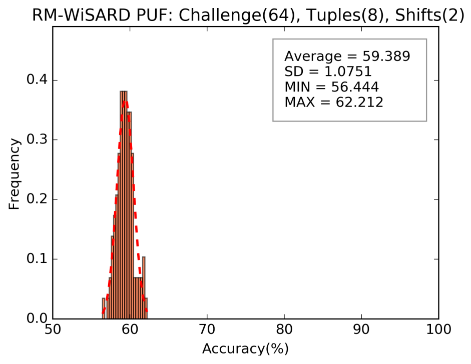
(d) WiSARD PUF with tuple rotation.



(e) RM-WiSARD PUF.



(f) RM-WiSARD PUF with extra bits.



(g) RM-WiSARD PUF with tuple rotation.

Figure 3.8: Gradient Boosting accuracy distribution.

main resource costs originate from the number of SRAM cells, area cost of final response bit generation and challenge-to-tuple mapping. The fixed mapping, illustrated in Figure 3.2, can be implemented with minimal resources and most designs require $\leq$ 2K SRAM bits aside from extra-bits schemes which require large number of SRAM bits, as shown in Table 3.1. The unit SRAM cell size of $0.346\mu m^2$, quoted by Intel [82], in 45 nm technology node consumes an area of $710\mu m^2$ for 2K SRAMs with additional area coming from interface circuitry. When IC is also utilized with WNN for neural network applications, their resource costs over both applications are amortized. Reed-Muller decoder hardware implementation detailed by Bösch [34] consumes $248.976\mu m^2$ area for $RM(1, 3)$ and the repetition code decoder 31.92 μm^2 area, both costs determined by 45 nm standard cell library [83]. These resources constitute a small overhead in comparison to the size of the RAM blocks and further reduction can be obtained by sharing decoder hardware and serializing the PUF operation.

3.3 Reliable Strong PUF Implementation

In this Section, an initial entropy source originated from reliable Weak PUFs is combined with Weightless Neural Networks (WNNs), using the WiSARD model, to construct reliable Strong PUF. Further, the basic WNN architecture, presented in Section 3.1, is modified with the purpose of improving the machine learning attack resistance of the Strong PUF. All presented architectures produce a 1-bit output response, given an m-bit input challenge.

3.3.1 Reliable Weak PUF Entropy Source

As discussed in Section 2.1.3, there are extensive solutions to make Weak PUFs reliable. To construct a reliable Strong PUF, Weak PUFs are utilized as the sources of entropy in the PUF system. By using potent Weak PUF designs [44], robust error correction mechanisms [41] and techniques such as accelerated aging and masking [40] enable to create a highly reliable binary string (or *secret key*) from an array of Weak PUFs. Such a binary string represents the process variation dependence of the final Strong PUF design and is unique to each IC.

In this work, the objective is to find the smallest length of the binary string required for creating a Strong PUF with high uniqueness and machine learning attack resistance. In addition, by figuring out the desired Strong PUF, it is possible to determine the smallest entropy source results to build the Strong PUF with smallest area. The initial reliable Weak PUF is set up to 256, 128, 64 or 32 bits that will be used to fill WNN RAM cells.

3.3.2 Complete Strong PUF architecture

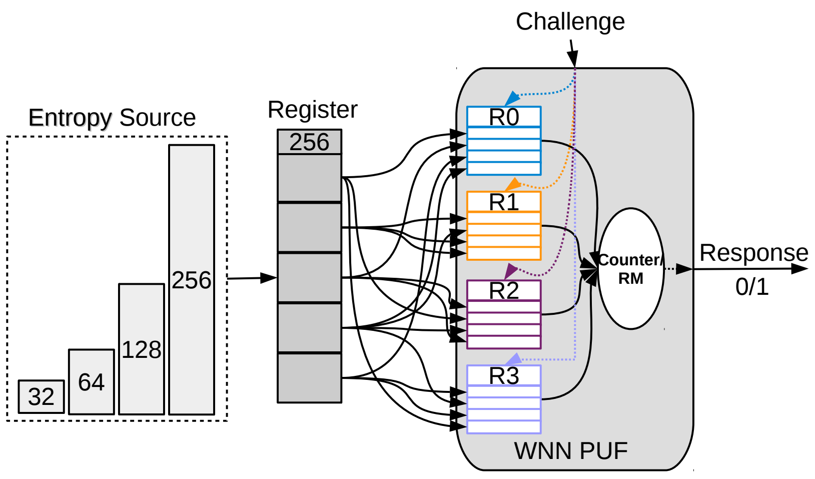


Figure 3.9: Reliable Strong PUF implementation [3].

The RAM contents are filled with the reliable Weak PUF bits in an equiprobable manner to reduce bias in the system. For example, if there are 2^{12} total RAM locations in the WNN and 2^7 reliable bits from Weak PUFs are obtained, then each bit needs to be randomly mapped to $2^{12}/2^7$ $(= 32)$ unique locations. The random mapping can be implemented in hardware by either hard-wiring each Weak PUF bit to the appropriate RAM cell or using a crossbar network.

The complete Strong PUF implementation, as illustrated in Figure 3.9, keeps a 256-bit register used to store the Weak PUF bits. The register itself receives its data from an initial number of reliable Weak PUFs, ranging from 32 to 256 bits. When the number of initial reliable bits does not complete the 256-bit register, the relevant number of copies are made to fill the total of 256 bits. Then, register bits are used to load the WNN RAM locations and the WNN processes the input challenges to produce the final 1-bit response.

3.4 Reliable Strong PUF - Experimental Setup and Results

The proposed reliable designs, presented in Section 3.3, are evaluated to figure out how the size of the initial *entropy source* in the form of reliable Weak PUF bits influences the machine learning resistance and uniqueness of Strong PUF. Lastly, the results are analyzed with the purpose of identifying the minimum entropy source size that can provide higher machine learning resistance and high uniqueness. Since in Section 3.3 the Weak PUFs have high reliability, all experiments consider that the architectures have 100% reliability.

3.4.1 Experimental Setup

The experiments for all PUF designs are simulated in Python and the relevant SRAM circuit parameters are obtained from SPICE simulations using 45 nm transistor models [80]. Each PUF applies a 64-bit input challenge and produces a 1-bit response. The Discriminator SRAM cell contents are filled according to the contents of the 256-bit register which, in turn, is loaded by the Weak PUF entropy source, as described in Section 3.3.2. The Weak PUFs in the entropy source are unique across PUF instances having equal number of 1's and 0's.

In this Section, the general details about WNN designs simulated are tabulated in Table 3.3. Note that the extra bits approaches were not included due to worst results discussed in Section 3.2. For designs that have 9 total RAM blocks, 7 are addressed by 8-bit tuples while the remaining two use 4-bit tuples. The designs with tuple rotations consider 2 rotations and the RM$(1,3)$ decoder is utilized for response generation by both RM-WiSARD PUF design variants. Finally, all the designs assume a fixed input-to-tuple mapping across PUFs, as depicted in Figure 3.2.

Table 3.3: Reliable WiSARD PUF Architectures with 64-bit challenges [3].

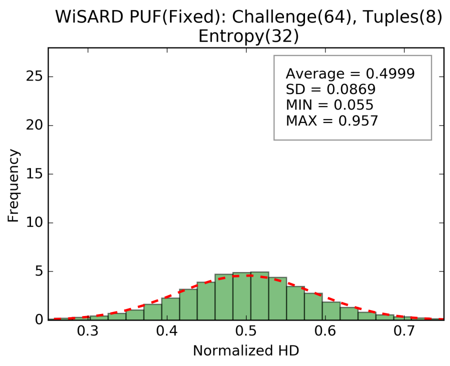
	#RAM Blocks	*# Addresses*	*# SRAMs*
WiSARD PUF	9	$7 \cdot 2^8 + 2 \cdot 2^4$	$1,824$
RM-WiSARD PUF	8	$8 \cdot 2^8$	$2,048$
WiSARD PUF + Tuple rotation	9	$7 \cdot 2^8 + 2 \cdot 2^4$	$1,824$
RM-WiSARD PUF + Tuple rotation	8	$8 \cdot 2^8$	$2,048$

3.4.2 Uniqueness

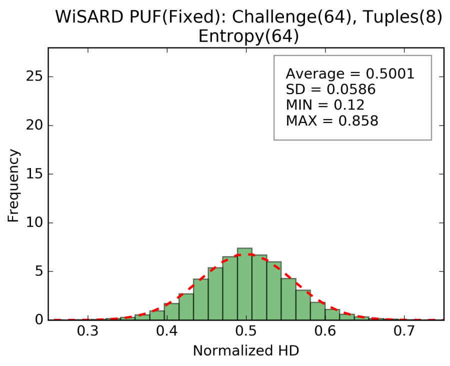
Section 3.2.2 defines uniqueness. For this experiment, 100 entropy source-to-WNN random mappings were generated for each PUF architecture. For each Strong PUF design, $(n =)$ 1000 challenges were applied for each one of the $(k =)$ 100 PUF instances. The entropy size was varied to 32, 64, 128 and 256 Weak PUF bits in each PUF instance, as shown in Figure 3.9.

The uniqueness results for different entropy source sizes are presented in Figure 3.10 and Figure 3.11, for WiSARD PUF versions, and Figure 3.12 and Figure 3.13, for RM-WiSARD PUF variants. These results were obtained by the *best* entropy source random mapping among the 100 generated mappings. For all designs and entropy source sizes, the mean of inter-class HD was closed to the ideal 0.5. The standard deviation decreases as the entropy source size increases in any given design.

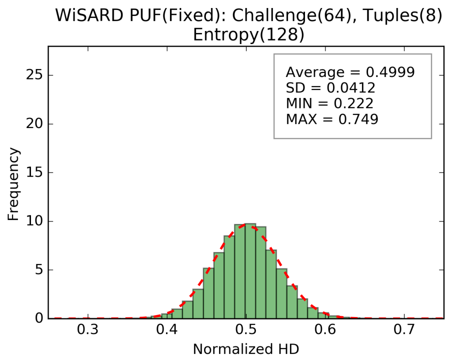
The original RM-WiSARD PUF with a 256-bit entropy source offered the lowest standard deviation with the mean inter-class HD of 0.4858.



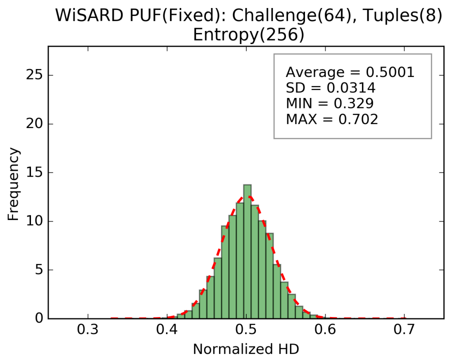
(a) Entropy size = 32.



(b) Entropy size = 64.



(c) Entropy size = 128.



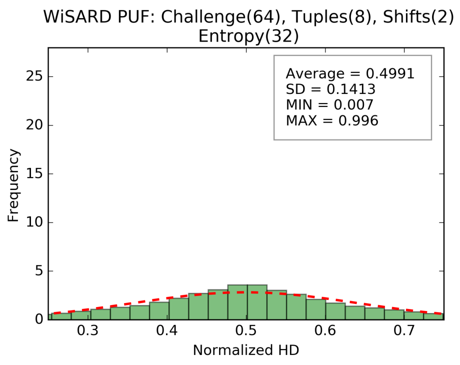
(d) Entropy size = 256.

Figure 3.10: Uniqueness distribution for WiSARD PUF with Fixed Tuples varying Entropy Source sizes.

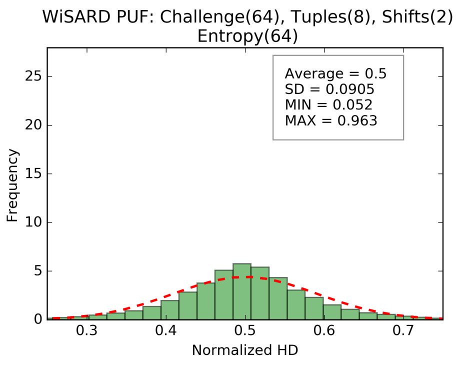
3.4.3 Machine Learning Resistance

Section 3.2.4 defines Machine Learning Resistance. In this experiment, only Gradient Boosting (Grad Boost) is used to estimate the attack resistance of the proposed reliable designs as it consistently outperforms LR and SVM.

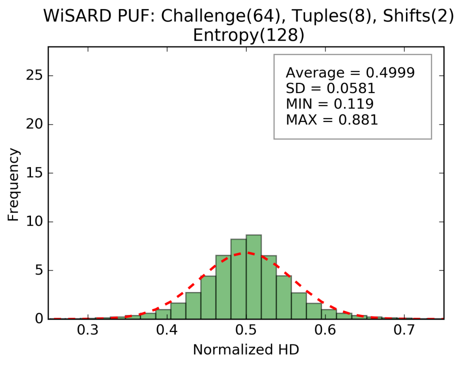
Gradient Boosting was implemented in Python using the scikit-learn tools [81] with the number of estimators set at 128 and learning rate of 0.01. For each scenario, the 10 best entropy source mappings were selected from the 100 mappings used by uniqueness results, representing the 10 smallest standard deviations. Thus, the number of experiments are reduced with the shortlist of the candidates for machine learning analysis. For each design, 100 PUF instances were simulated with 150, 000 CRPs, where 100, 000 CRPs were utilized for *training* a model with Gradient Boosting and 50, 000 CRPs were used to test the model in order to measure the machine learning accuracy of the cloned PUF.



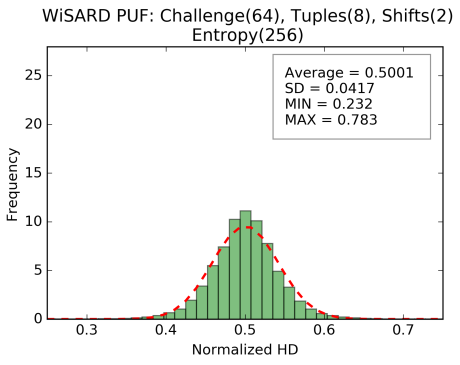
(a) Entropy size = 32.



(b) Entropy size = 64.

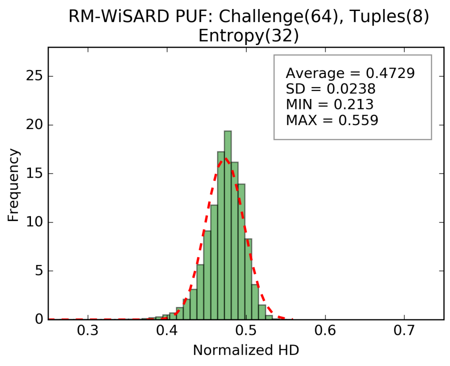


(c) Entropy size = 128.

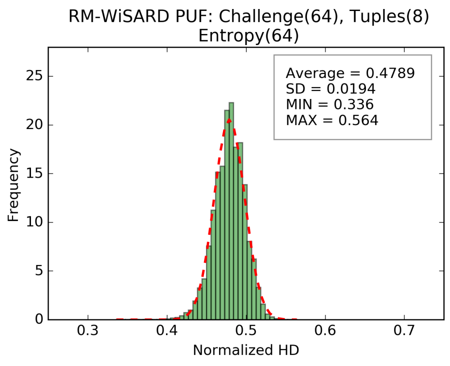


(d) Entropy size = 256.

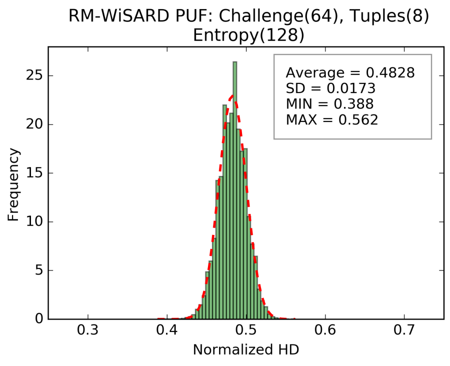
Figure 3.11: Uniqueness distribution for WiSARD PUF with tuple rotation varying Entropy Source sizes.



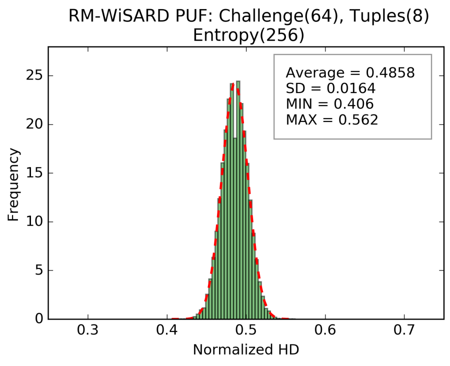
(a) Entropy size = 32.



(b) Entropy size = 64.

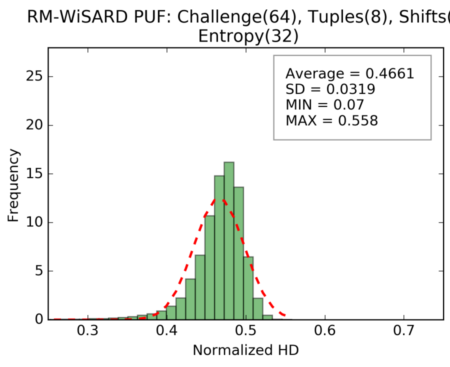


(c) Entropy size = 128.

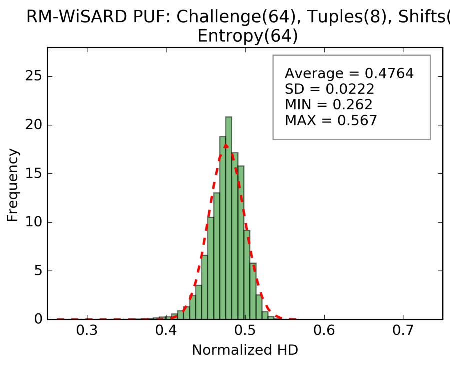


(d) Entropy size = 256.

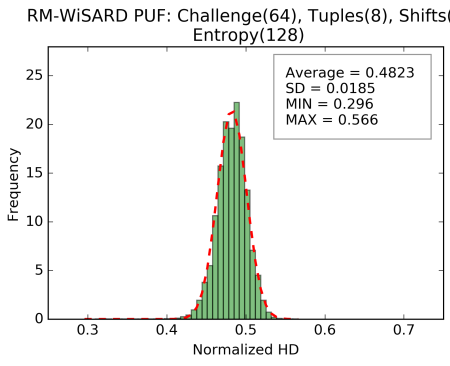
Figure 3.12: Uniqueness distribution for RM-WiSARD PUF varying Entropy Source sizes.



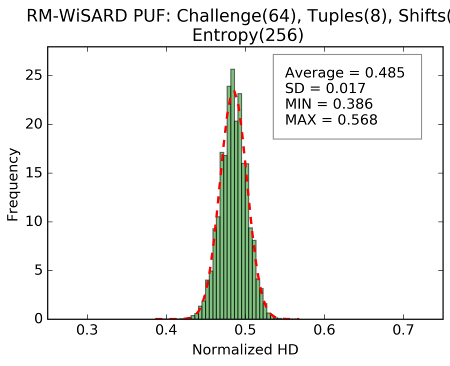
(a) Entropy size = 32.



(b) Entropy size = 64.



(c) Entropy size = 128.



(d) Entropy size = 256.

Figure 3.13: Uniqueness distribution for RM-WiSARD PUF with tuple rotation varying Entropy Source sizes.

The average machine learning accuracy for each PUF architecture with varying entropy source sizes is summarized in Table 3.4. The results consider the best mapping of the 10 shortlisted entropy source mappings. Figure 3.14 and Figure 3.15 present the distributions of machine learning accuracies for varying entropy sizes for both RM-WiSARD PUF implementations. These implementations provided the best resistance results (lower machine learning accuracies) compared to the other architectures, with the best results offered by RM-WiSARD design. Similarly to the uniqueness results, when the entropy source size increases, the machine learning accuracy and standard deviation decrease. The random entropy mapping is chosen for each design by considering the results from an entropy source size of 256 bits and then, changing the size while keeping the same mapping. RM-WiSARD PUF variants offered higher machine learning resistance than the simple WiSARD variants. Furthermore, it is possible to reach $\leq 65\,\%$ machine learning accuracy by considering just 32 initial reliable Weak PUF bits. Therefore, the RM-WiSARD PUF with an entropy size of 32 can offer high machine learning resistance while still exhibiting good uniqueness. That fact allows to realize a smaller Strong PUF.

Table 3.4: Gradient Boosting-based Machine Learning Accuracy for WiSARD PUF variants [3].

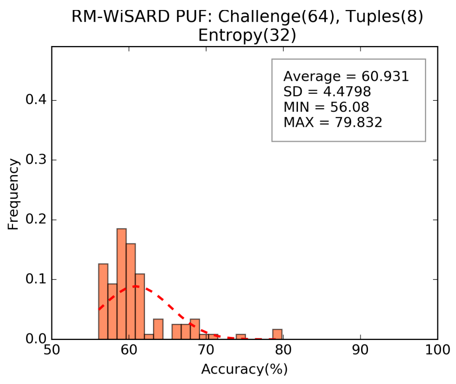
PUF Type	**Machine Learning Accuracy (%)**			
	32	**64**	**128**	**256**
WiSARD PUF	82.60	81.87	81.74	81.26
RM-WiSARD PUF	60.93	60.17	59.39	59.00
WiSARD PUF + Tuple rotation	85.22	83.40	82.32	81.74
RM-WiSARD PUF + Tuple rotation	64.87	62.44	60.93	59.15

3.5 Concluding Remarks

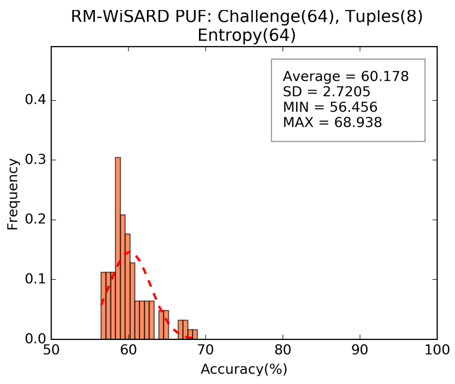
Attack Scenario

In this work, the attacker can only intercept the PUF CRPs for use in model building attacks, in other words, the PUF is considered as *black box*. The designer has to take the necessary precautions such as: ensure that RAM block contents, required in the neural network implementation, are not accessible outside the system and prevent any data leakage from PUF system, especially the Weak PUF bits.

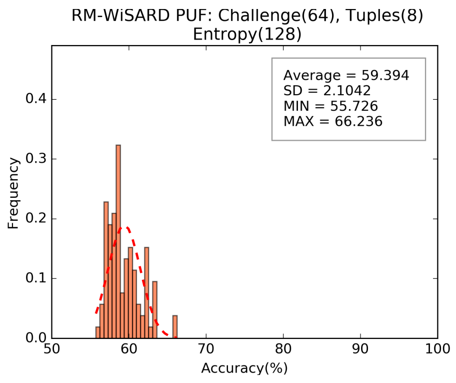
Further, the black box perspective provides protection against brute-force guessing of possible bit values, even utilizing just 32 initial reliable Weak PUF bits. Indeed, a designer can increase security by choosing a larger entropy source. Future works will focus on studying the security of the PUF in case an attacker has knowledge of the PUF architecture due to possession of a physical device. Possible fault injection attacks will also be explored.



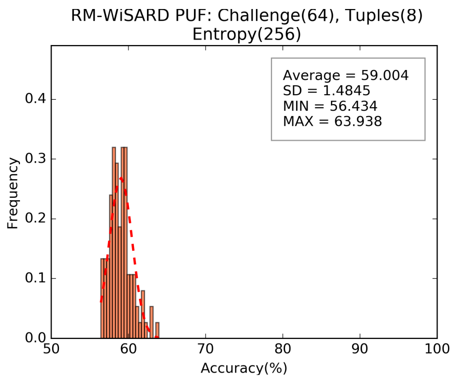
(a) Entropy size = 32.



(b) Entropy size = 64.

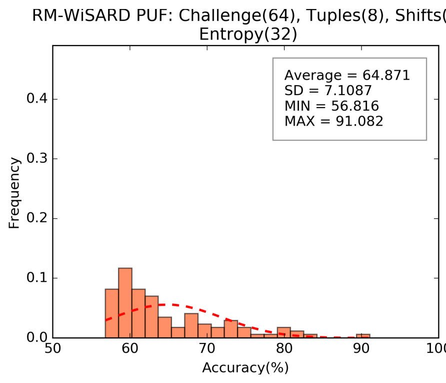


(c) Entropy size = 128.

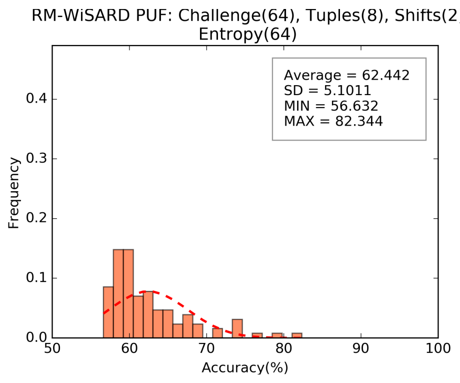


(d) Entropy size = 256.

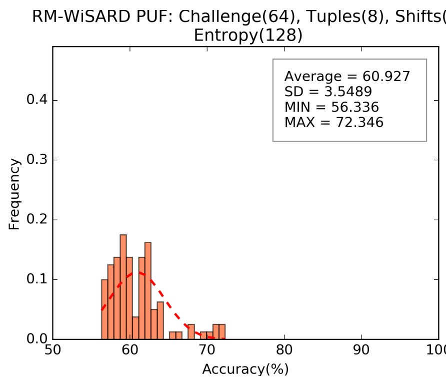
Figure 3.14: Gradient Boosting machine learning accuracy distributions for RM-WiSARD PUF.



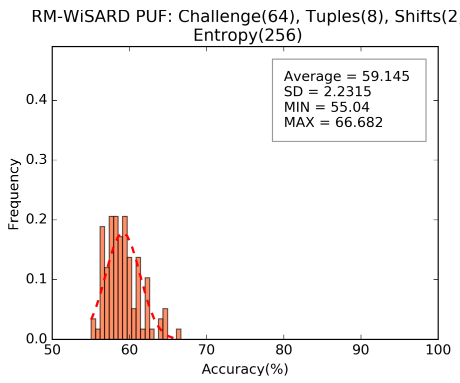
(a) Entropy size = 32.



(b) Entropy size = 64.



(c) Entropy size = 128.



(d) Entropy size = 256.

Figure 3.15: Gradient Boosting machine learning accuracy distributions for RM-WiSARD PUF with tuple rotation.

Hardware Implementation

The major resource costs for the reliable designs come from the number of WNN memory cells required. As presented from Table 3.3, the WNN implementations require a maximum of 2K bits and the details of hardware resources are discussed in Section 3.2.5.

The Weak PUF block required to generate the reliable bits for the WNN is other area intensive unit. As demonstrated from Figure 3.12 and Table 3.4, the RM-WiSARD PUF architectures offer both high uniqueness and machine learning resistance by considering just 32 initial reliable Weak PUF bits. Considering the details provided by Vijayakumar *et al.* [41] and assuming a 45 nm standard cell library [83], the final reliable 32 bits can be obtained by using 16 : 1 mux implementation with a 5 % observable yield loss that necessitates 48 ($\in 16\mathbb{Z}$) cells. This results in area of $\approx 550\mu m^2$, which is significantly less than the area that the WNN memory cells need.

The use of keyed hash functions is unfeasible with Weak PUF bits to obtain a Strong PUF implementation, as Keccak [84]. The area for an efficient keyed hash implementation was found to be 2280 gate-equivalents [84] or $\approx 1900\mu m^2$ in 45 nm standard cell library [83]. As discussed in Section 3.2.5, the majority of the area in the design is dependent on the RAM implementation with 2K SRAM unit cells occupying $710\mu m^2$ in 45 nm. The area of PUF is efficient compared to the keyed hash function.

The experiment results assumed the best random mapping between the Weak PUF bits and WNN. However, other random mappings can yield similar favorable results in terms of machine learning resistance. Hence, it might be possible for a designer to analyze multiple such mappings and incorporate them into a design, with the ability to choose one among them. The WNN mapping can be built dependent on the Weak PUF bit values which are unique to each IC. It implies further enhance the Strong PUF security by foregoing a fixed mapping across all ICs. That circuitry will be more costly than the hard-wired option.

Table 3.5: Combinational Logic implementations of PUF with varying Entropy Sources.

Entropy Source (bits)	Area (μm^2)
32	1710
64	2060
128	2400
256	2650

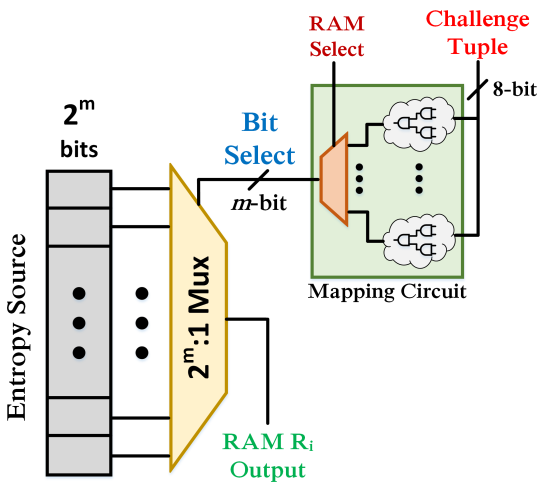


Figure 3.16: Combinational logic-based implementation of Strong PUF [3].

Alternatively, PUF implementation can be based on two mapping operations. The first mapping is between the input challenge to the tuple for addressing the RAM blocks. The second mapping is for the RAM content based on the Weak PUF entropy source. These maps can be implemented as a 2D crossbar or as combinational logic. Figure 3.16 shows the schematic for a combinational system. The *RAM Select* is used for virtual RAM block (R_i) selection while the specific bit is selected via *Bit Select* control. Table 3.5 summarizes the silicon areas, in 45 nm standard cell library [83], for varying sizes of the entropy source. One observation is that the *randomly mapped* implementation of the proposed Strong PUF compares favorably with the *optimized* Keccak design.

Conclusion

Strong PUF is promise to provide a low cost alternative to cryptography-based authentication. However, the Strong PUF implementations suffer security issues so that model building attacks using machine learning techniques can clone the circuit with high accuracy. In addition, their unreliability demands the increase on the number of CRPs to authenticate properly. Hence, there is still need to implement a Strong PUF architecture that provides high reliability and greater resistance to machine learning attacks.

In this work, two issues were addressed. Former, a novel Strong PUF architecture integrating WiSARD Neural Network (WNN) hardware was proposed to increase machine learning attack resistance and, further, extensions of the original design were explored. The promising results demonstrate as neural networks could be viable to create Strong PUF candidates achieving high resistance against machine learning attack, uniqueness and reliability with low resource overhead. Latter, an initial reliable Weak PUF entropy source was mapped into WNN to construct reliable Strong PUFs. The minimum entropy source analysis shows that it is possible to create highly reliable Strong PUFs with $< 65\,\%$ ML accuracy by using as few as 32 reliable Weak PUF bits.

Chapter 4

Efficient Testing Strong PUF for Uniqueness

As introduced in Section 2.1, PUFs promise unclonability property through high uniqueness that relies on manufacturing process variations. Nevertheless, that property cannot be guaranteed only by design since the internal components are not controllable and, consequently, two PUFs might have identical characteristics. Therefore, testing manufactured PUFs is an essential process to find out the circuits that failed to provide the desired property. Although the problem of testing PUF for uniqueness is critical, it has not received much research attention.

In a population of PUFs, high uniqueness can be trivially ensured by collecting their responses and performing an offline comparison. This method requires large resource costs as chips are tested several times over testing production line on multi-socketing. Systematic methods and metrics to analyze PUF characteristics have focused on design time evaluation [85]. To evaluate security of PUF, Majoobi *et al.* have proposed techniques that have reported inadequate security of various PUFs such as linear PUFs and feed-forwards PUFs [86]. These techniques and metrics can be implemented for multi-socketing and are suitable for benchmarking and offline evaluation during design phase or pre-high volume manufacturing phase. In High Volume Manufacturing (HVM) testing, the decision to accept or reject chips should be performed in real-time at the tester. Thus, to adapt above mechanisms in HVM, it would be required to use additional steps to discard non-unique chips resulting in extra cost.

Built-in-self Test (BIST) based schemes have been explored to test Fuzzy Extractor [87], which is one of the main part of memory-based PUF, and to evaluate unpredictability and reliability of PUFs [88]. However, these techniques have not considered the problem of testing PUF for uniqueness. Recently, Vijayakumar *et al.* [89] have proposed techniques for testing uniqueness based on design-for-test (DFT) methods tailored for high-volume testing. Strong PUF testing mechanism performs a pass/fail decision for each new chip by comparing its responses to the similar responses of passed chips shortlisted from a multi-index hashing (MIH) structure introduced in Section 2.5. MIH is maintained to fast search the nearest neighbor in all passed chips responses in order to achieve an efficient run-time and acceptable rates of false-accepted and yield-loss.

As Strong PUFs are practical solutions to implement low-authentication in IoT applications, millions or billions of PUFs can be manufactured by a single manufacturing facility. Uniqueness testing utilizing MIH can suffer from extremely high memory cost as it requires to maintain the entire database of responses in main memory to accurately decide the rejection or acceptance of a chip. In addition, the memory cost increases to maintain the MIH structure which replicates a part of response into multiple hash tables. Motivated by the Strong PUF testing proposed by Vijayakumar *et al.* [89], this Chapter explores MIH implementations and investigates online solutions for nearest neighbor search in Hamming space problem with the aim at reducing the storage space while keeping an efficient performance.

4.1 MIH for Testing Strong PUFs

In this Section, the problem of uniqueness testing for Strong PUFs is formulated. Next, the methodology proposed by Vijayakumar *et al.* [89] is detailed.

4.1.1 Metrics for Uniqueness

Uniqueness is calculated by Inter-class Hamming distance (HD) metric that represents the differences among the responses produced from various PUF instances when they receive the same challenge. The average of inter-class HD, d_{inter}, is calculated considering the HD results from each pair of PUFs over many challenges according to the following formula:

$$d_{inter} = \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{HD(r_i, r_j)}{n} \tag{4.1}$$

where k is amount of PUFs, n is the total of bit responses and $HD(r_i, r_j)$ is Hamming distance between responses of the PUF instances i and j to a particular challenge.

4.1.2 Problem Statement

Considering a manufacturer has to test N manufactured PUFs for uniqueness, let us suppose that N_{passed} is the number of chips tested and passed by the tester and n-bit responses were obtained from each chip. Let R_i be the n-bit response from current

PUF i under test. The problem is defined as: (i) how to compare the current R_i to the N_{passed} responses at real-time and (ii) how to realize pass/rejection decision for uniqueness based on responses.

To exemplify, let us assume that every time the tester extracts 1000-bit response and compares it to N_{passed} responses stored in a database. One straightforward solution would be to run a linear scan in the database comparing the current response R_i to each one of N_{passed} responses which would take $O(n)$ run-time. This run-time can be prohibitively long as the number of manufactured chips reaches in millions.

Other solution would be to use hash table/dictionary based storage for efficient exact pattern matching. For example, let us suppose that a chip will be rejected if its response differs at most 10% HD, that is 100 bits, from any of N_{passed} responses. It requires to generate all the combinations differing at most 100 bits and execute $\binom{1000}{100} \approx 10^{139}$ look-ups which is infeasible for real-time constraint.

4.1.3 Multi-Index Hashing for Testing PUF

MIH is used to reduce the search time by minimizing the number of passed chips verified with the current chip for testing uniqueness.

An example of MIH search with 5 hash tables for the problem of testing PUF uniqueness is detailed below. Let us consider a 100-bit response of a chip is received to find all neighbors that are at 10 Hamming distance from it. First, 100-bit response is divided into five 20-bit sub-strings which will be used as key for each hash table. According to equation 2.1, if two 100-bit responses differ by at most 10 bits, then at least one of the corresponding sub-strings differ at least by $10/5 = 2$ bits. So, all 5 hash tables are searched for neighbors at radius $r = 2$ from each 20-bit sub-string. As result, the number of lookups is decreased from $\binom{100}{10} \approx 1,7 \times 10^{13}$ to $5 \times \binom{20}{2} = 950$. Then, each neighbor of the sub-string with value in the hash table has to be verified with the entire 100-bit response to confirm whether it is a true 10-distant neighbor or not.

4.1.4 Uniqueness Test Procedure

As reported in [89], the practical number of CRPs for tester evaluation is determined in 1000 bits and the decision threshold is defined in 10%. In other words, a chip is considered identical if its 1000-bit response differs at most 100 bits (10%) from any passed chip. These parameters were obtained by empirical simulations.

The procedure for testing uniqueness is illustrated in Figure 4.1. The tester keeps a database with 1000-bit response of all passed chips. During testing, each chip produces 1000-bit response and sends to the tester. A pass/fail decision is performed by comparing the candidate responses selected from MIH search with the

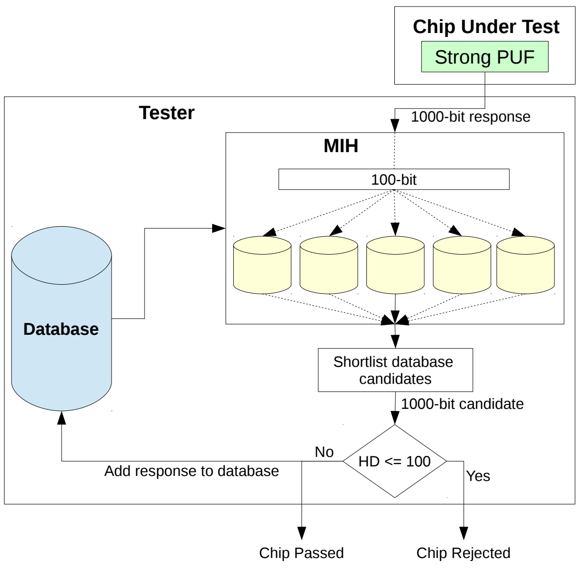


Figure 4.1: Example of testing Strong PUF. MIH has 5 hash tables and the neighbors are selected by searching at radius $r = 20$.

current chip considering the decision threshold. The first 100 bits of the response are used by MIH to find all neighbor candidates with similar responses at radius 20, as exemplified in Section 4.1.3. As MIH has 5 hash tables, each 20-bit sub-string is searched at radius $r = 20/5 = 4$ in each hash table. Although using 10% threshold indicates the MIH radius $r = 10$, the radius $r = 20$ is applied to minimize the *yield loss* (a good chip is rejected) and *false-accepts* (a bad chip is accepted) of chips. At the end, the found candidates are evaluated using the complete 1000-bit response to confirm the true HD. If HD from any candidate response is lower the decision threshold (HD $\leq$ 100), then the current chip is rejected. Otherwise, the chip is accepted and its response is added to the database.

4.2 Analyzing MIH Implementations

This section details MIH implementations applied to the testing procedure for Strong PUF explained in Section 4.1.4. First, the adaptation of original MIH implementation [76] to testing procedure is discussed. Then, an optimized implementation is elaborated and evaluated.

4.2.1 Original MIH Implementation

As introduced in Section 2.5, the original MIH copies a binary code into m hash tables indexed by one of its m sub-strings. Figure 4.2 illustrates the testing procedure implemented with original MIH. Since 100-bit of Strong PUF response is used by MIH with 5 hash tables, 20-bit key maps a list of 100-bit values (or the remaining 80

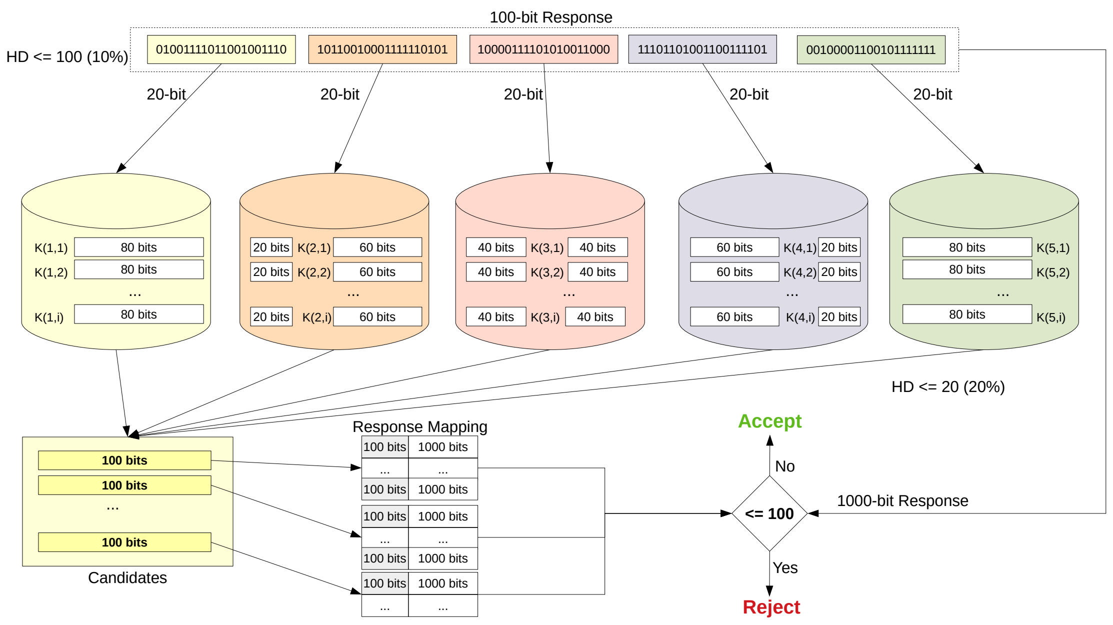


Figure 4.2: Example of testing Strong PUF using original MIH.

bits) in each hash table. During MIH searching, 0-4-distant neighbor keys from each corresponding sub-string are looked up and all matched 100-bit values are evaluated with 100-bit query. If HD ≤ 20 then the matched 100-bit value is selected as candidate for the last step which verifies the HD decision threshold ≤ 100 with the entire 1000-bit response. For this last verification, a response mapping table is required to map 100-bit candidates to 1000-bit responses already stored in the database.

4.2.2 MIH Implementation with Global Index

The original MIH implementation, presented in Section 4.2.1, requires a response mapping table to obtain the 1000-bit responses pointed out by a 100-bit candidate. As the database already stores 1000-bit response, this table can be removed if the global index of the response stored in the database is held by MIH. The modified MIH copies a global index into 5 hash tables, instead of copying 100 bits as depicted in Figure 4.3. The global index length can be setup to 32-bit that is enough to create indexes for responses of billions tested chips stored in the database. During the MIH search, all global indexes looked up by the neighbor keys in each hash table are used to access the 100-bit response from database. Similar to original MIH, the 100-bits are evaluated to select the candidates for the last verification. As the candidates are also global indexes, the 1000-bit neighbors are directly accessed from database and compared to the query for finally making the pass/fail decision.

This optimization eliminates the response mapping table and reduces the value length from 100 bits to 32 bits stored for each hash table.

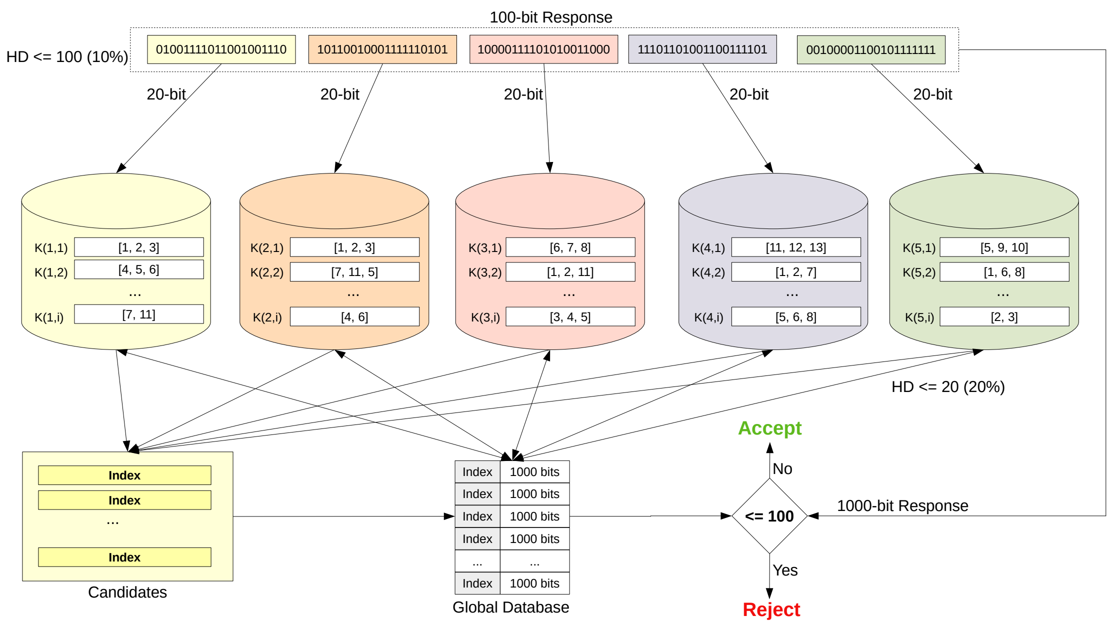


Figure 4.3: Example of testing Strong PUF using MIH with global index.

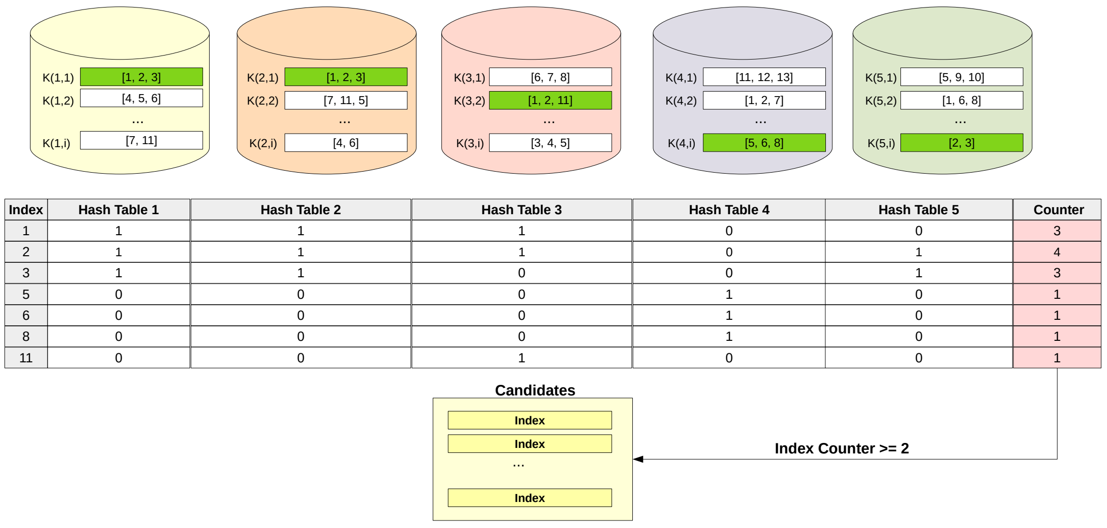
4.3 Strategies for Memory Reduction

The MIH implementations discussed in Section 4.2 require to maintain the PUF responses in main memory for pass/fail verification with 1000 bits. As the database grows rapidly, the memory cost may become impractical. This Section defines two strategies to mitigate the memory cost.

4.3.1 Distance Free Computation Strategy

The distance free computation strategy replaces the HD calculation used to select the candidates with HD ≤ 20 during MIH searching. The idea is based on distance-computation-free search scheme proposed by Song *et al.* [90], which have observed that HD is naturally embedded in the inverted multi-index structure like MIH. Since MIH with global index copies each index i to m hash tables, a query is very similar to the response pointed out by index i if almost all matched hash table locations have stored i. Case all m hash tables return the same index i, so it means that the query has already been inserted into MIH.

An example is shown in Figure 4.4. A query is searched on MIH with 5 hash tables and the highlighted (green) hash table locations are matched by the neighbors of each sub-string. Instead of using the indexes to access the 100-bit responses for HD calculation, the accessed indexes are counted across the hash tables. The index counter varies from 0 to 5 (number of hash tables) and the closer to 5 it is more similar to the query. In the example, index 2 is more identical as its index counter is 4, that is, 4 sub-strings are equal to the query. Therefore, the similarity can be



Index	Hash Table 1	Hash Table 2	Hash Table 3	Hash Table 4	Hash Table 5	Counter
1	1	1	1	0	0	3
2	1	1	1	0	1	4
3	1	1	0	0	1	3
5	0	0	0	1	0	1
6	0	0	0	1	0	1
8	0	0	0	1	0	1
11	0	0	1	0	0	1

Figure 4.4: Example of distance free computation strategy.

established by defining an index counter threshold based on the bitmatrix idea used in [91]. The indexes are selected as candidates if their index counter are greater or equal than the index counter threshold (e.g. ≥ 2).

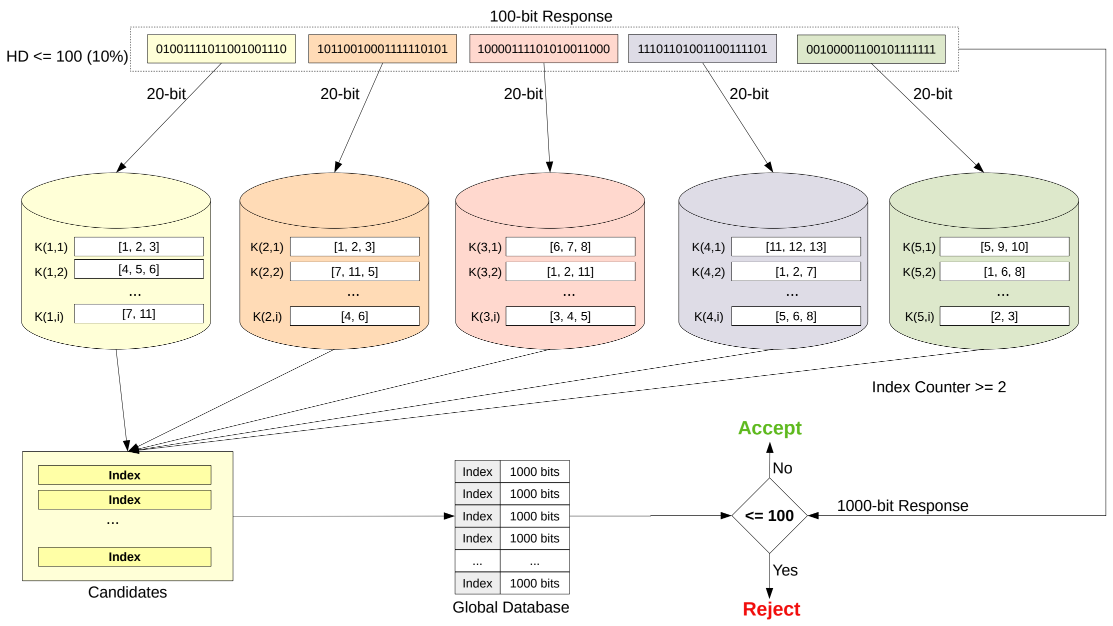


Figure 4.5: Example of MIH using distance free computation strategy. An index counter threshold is used to select the neighbor candidates.

By using distance free computation strategy, the database is not used during MIH searching as illustrated in Figure 4.5.

4.3.2 Hamming Weight Strategy

The hamming weight strategy creates a compact representation of the responses stored in the database. Hamming Weight (HW) of a binary code is defined as number of 1's. The idea is based on the statement formulated by Eghbali *et al.* [92]: when two binary code g and h differ by at most r bits then the difference between their Hamming weights is at most r. This statement establishes a relationship between HD and HW and, thus, the HD calculation can be replaced by HW differences calculation. By calculating the difference d of HW from two binary codes, they are most likely identical if d tends to zero. Considering a binary code $g = 11110000$ ($HW = 4$), a 1-distant neighbor of g has $HW = 4 - 1 = 3$ (11100000) or $HW = 4 + 1 = 5$ (11110001). Applying the same logic, the 2-distant neighbor of g has $HW = 4 - 2 = 2$ (11000000) or $HW = 4 + 2 = 6$ (11110011). So, HW difference $d = 1$ indicates the binary codes are likely 1-distant neighbors, $d = 2$ indicates they are likely 2-distant neighbors and so on. Note that HW difference n does not guarantee two binary codes are exact n-distant neighbors such as binary code $h = 00000111$ ($HW = 3$) is not 1-distant neighbor of g but 7-neighbor. The strategy consists of splitting a 1000-bit response into p parts with s bits and store only the integer which represents the HW from each part. As the HW varies from 0 to s, the integer requires $\lceil \log_2 s + 1 \rceil$ bits.

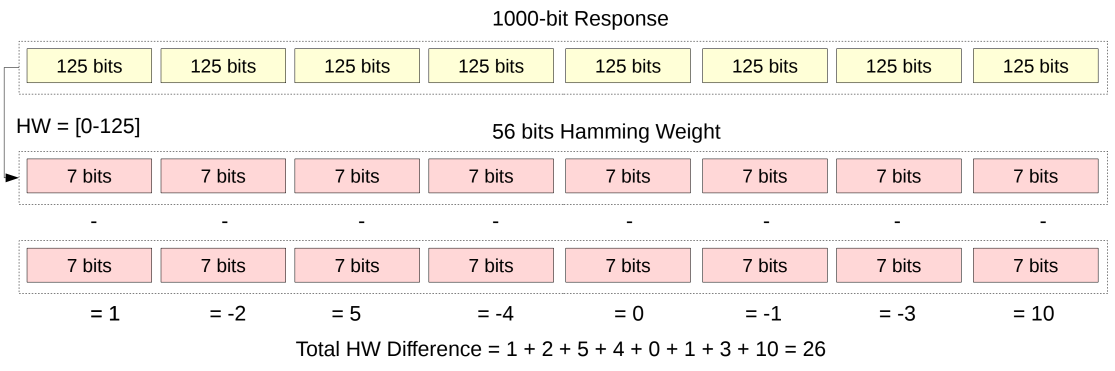


Figure 4.6: Example of hamming weight strategy. A 1000-bit response is split into 8 parts of 125 bits where the HW requires 7 bits resulting in total 56 bits.

An example of hamming weight strategy is presented in Figure 4.6. A 1000-bit response is divided into 8 parts of 125 bits where each part has HW in range 0-125 represented by 7 bits. Consequently, the response representation is reduced from 1000 to $8 \times 7 = 56$ bits. By using HW partitions of two responses (query and candidate), the pass/fail decision is making by comparing if the sum of the absolute HW differences (in the example is 26) is less or equal than a HW difference threshold.

Figure 4.7 depicts MIH using hamming weight strategy. The hamming weight

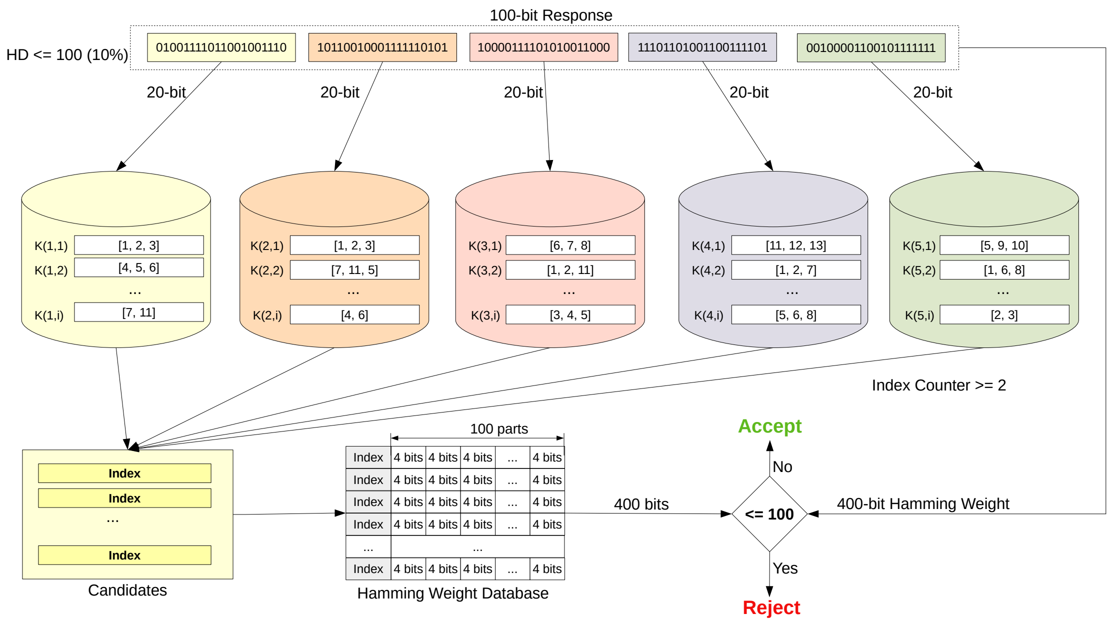


Figure 4.7: Example of MIH using hamming weight strategy.

database is maintained in main memory instead of database with 1000-bit responses. Similar to HD calculation, the total HW differences are calculated and evaluated with the decision threshold in order to decide if chip is rejected or accepted.

4.4 Experiments and Results

4.4.1 Experimental Setup

To compare the effectiveness of the MIH implementations and strategies discussed in Section 4.2 and 4.3, the experiments of quality test, memory consumption and performance are performed in 10 dataset varying the number of PUFs from 100, 000 to 1, 000, 000. Each dataset was created to contain 1000-bit PUF responses with an average of HD of 10%, that is, half of the chips will be rejected during testing. Both MIH structures were implemented in C and MIH search is executed in parallel using 5 threads (one for each hash table). The experiments were performed on an Intel Xeon E5-2640(2.6GHz) processor and 128GB of RAM.

4.4.2 Original MIH vs MIH with Global Index

The original MIH and MIH with global index implementations are evaluated. For each dataset, the linear scan search was performed to generate the acceptance and rejection results which will be used to compare with MIH results.

The quality test results are presented in Table 4.1. The yield loss is expressed by False Rejected (FR) rate (chips should be accepted but were rejected) and the faulty chip is indicated by False Accepted (FA) rate (chips should be rejected but were accepted). The results are the same for both MIH implementations where all datasets obtained low yield loss and faulty chip accepted.

Table 4.1: MIH quality test results. TR = True Rejected, FR = False Rejected, TA = True Accepted, FA = False Accepted.

# Resp	**TR %**	**FR %**	**TA %**	**FA %**
100K	49.977	0	50	0.23
200K	49.983	0	50	0.017
300K	49.98	0	50	0.02
400K	49.982	0	50	0.18
500K	49.982	0	50	0.018
600K	49.979	0	50	0.021
700K	49.982	0	50	0.018
800K	49.979	0	50	0.02
900K	49.983	0	50	0.016
1M	49.983	0.0001	50	0.017

Table 4.2: Performance and memory consumption for original MIH.

# Resp	**Init Time (s)**	**Testing Time (s)**	**MIH Memory (MB)**	**Total Memory (MB)**
100K	0.047	36.477	23.82	30.07
200K	0.059	88.419	27.63	39.99
300K	0.06	162.004	31.45	49.92
400K	0.08	250.8644	35.26	59.84
500K	0.085	354.9	39.08	69.76
600K	0.106	468.659	42.9	79.69
700K	0.118	617.834	46.71	89.6
800K	0.13	783.736	50.53	99.53
900K	0.143	965.167	54.34	109.45
1M	0.154	1133.502	58.16	119.37

For each dataset, the experiments were run 10 times and the average of the performance was calculated. Table 4.2 and 4.3 show the performance and memory consumption results for original MIH and MIH with global index, respectively. The init time encompasses the initialization of database, MIH and response mapping table, while the testing time comprises the testing procedure computation. The total memory includes the MIH memory and response mapping table resources. MIH with global index considerably reduced the total of memory compared to original MIH by eliminating the response mapping table and reducing the hash table values from 100 to 32 bits. Also, it achieves comparable performance to the original MIH.

Table 4.3: Performance and memory consumption for MIH with global index.

# Resp	Init Time (s)	Testing Time (s)	MIH Memory (MB)	Total Memory (MB)
100K	0.014	36.203	20.95	20.95
200K	0.024	95.027	21.91	21.91
300K	0.029	171.814	22.86	22.86
400K	0.034	269.687	23.82	23.82
500K	0.05	393.606	24.77	24.77
600K	0.061	557.314	25.72	25.72
700K	0.07	690.128	26.68	26.68
800K	0.075	876.272	27.63	27.63
900K	0.076	1089.483	28.58	28.58
1M	0.092	1347.237	29.54	29.54

4.4.3 Strategy Thresholds Evaluation

The strategies presented in Section 4.3 rely on some threshold parameters to progress the execution. To find the better threshold values, various quality test experiments were run using different configurations.

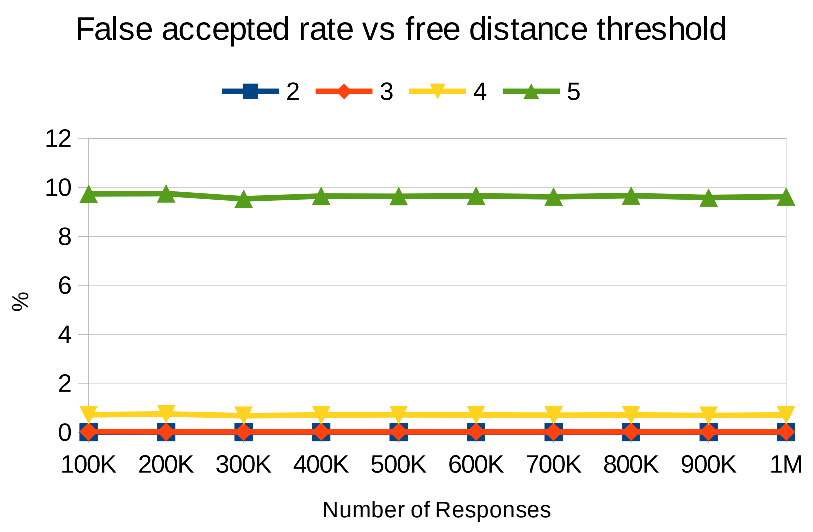
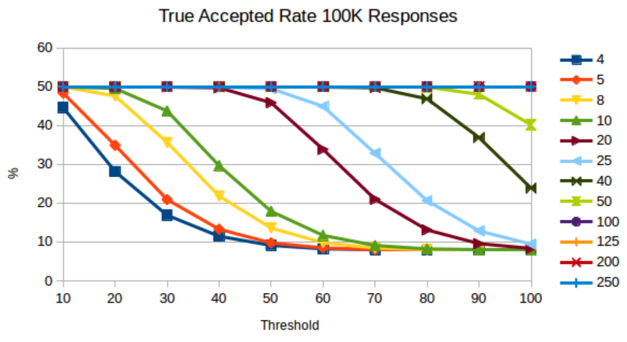
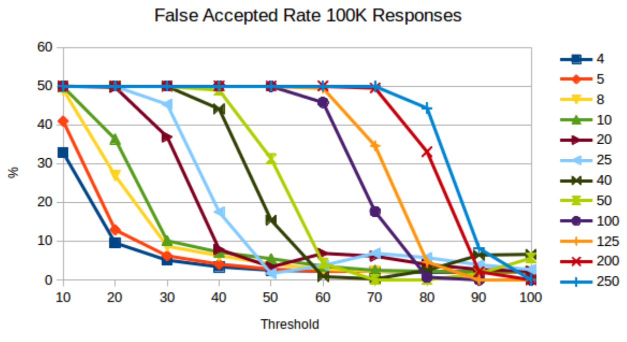


Figure 4.8: Faulty chip rate for different configurations using distance free computation strategy. Each line represents one threshold configuration.

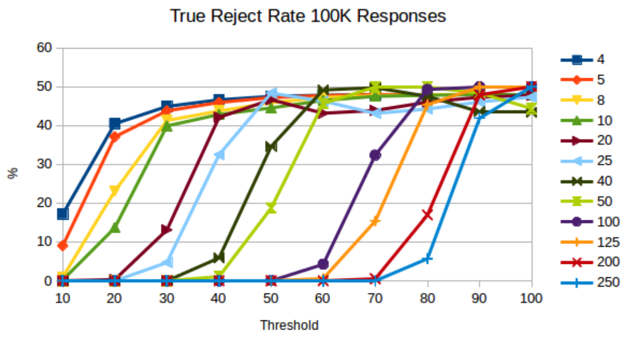
As discussed in Section 4.3.1, the distance free strategy utilizes the index counter threshold to select a matched neighbor from MIH as candidate. The index counter threshold was setup with values between 2 and 5 in all datasets. The results of faulty chip rate are shown in Figure 4.8. The threshold configurations to 2 and 3 achieved low false-acceptance ratio. The index counter threshold = 2 was chosen to next experiments due to it results in lower rates.



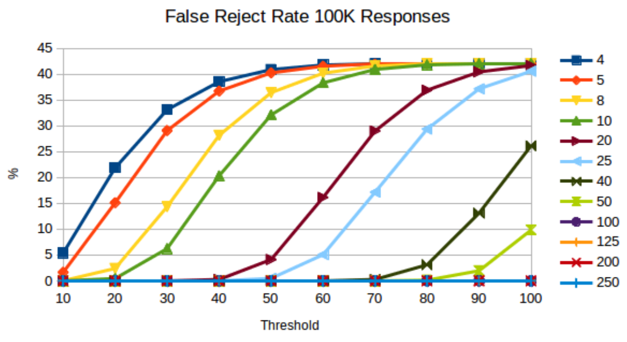
(a) True accepted rate (TA). The closer to 50% the better.



(b) False accepted rate (FA). The closer to 0% the better.



(c) True rejected rate (TR). The closer to 50% the better.



(d) False rejected rate (FR). The closer to 0% the better.

Figure 4.9: Experiment of true/false rates vs partition threshold. Each line represents a partition configuration and the thresholds vary from 0 to 100.

Table 4.4: The best HW difference thresholds per partitions using 100K responses. The best threshold is selected from 10 to 100 interval. The Bits column is the number of bits per partition and MEM is the total of bits required for all HW partitions.

# Parts	Bits	Threshold	TA%	FA%	TR%	FR%	MEM (bits)
4	250	20	28.12	9.53	40.47	21.88	32
5	200	20	34.89	12.89	37.11	15.11	40
8	125	30	35.65	8.69	41.31	14.35	56
10	100	40	29.66	7.15	42.85	20.34	70
20	50	40	49.73	7.87	42.13	0.27	120
25	40	50	49.51	1.64	48.36	0.48	150
40	25	60	49.99	0.83	49.17	0.01	200
50	20	70	50.00	0.06	49.94	0.00	250
100	10	$90, 100$	50.00	0.00	50.00	0.00	400
125	8	100	50.00	0.00	50.00	0.00	500
200	5	100	50.00	0.00	50.00	0.00	600
250	4	100	50.00	0.00	50.00	0.00	750

For hamming weight strategy, the HW difference threshold is used to make the pass/fail decision to reject or accept the chip as described in Section 4.3.2. To find the number of partitions and threshold that obtain better results, the experiments were performed with different configurations of number of partitions and thresholds in the dataset with $100,000$ PUFs. For each scenario, the threshold varies from 0 to 100.

Figure 4.9 shows the acceptance (FA, TA) and rejection (FR, TR) rates for different number of partitions and thresholds. The best results were achieved by number of partitions above 50, where false rates are closer to 0% and true rates are closer to 50%. The best thresholds per number of partition are summarized in Table 4.4. From 50 partitions, the results reach the ideal rates (false = 0% and true = 50%). The total memory increases as long as the number of partitions increases. Thus, the better parameters are 100 partitions with 100 threshold replacing 1000-bit response to 400 bits.

4.4.4 MIH Versions Analysis

Considering the best thresholds obtained for each strategy in Section 4.4.3, five versions of MIH were evaluated: original MIH (MIH), MIH with global index (MIHI), MIH with global index and distance free computation strategy (Free MIHI), MIH with global index and hamming weight strategy (HW MIHI), and MIH with global index and both strategies (Free HW MIHI). Each scenario was run 10 times.

The quality test results are tabulated in Table 4.5. Overall, all MIH versions offered low yield loss and faulty chip acceptance in datasets with low uniqueness.

Table 4.5: Quality test results for all MIH versions.

# Responses	**Rates**	**MIH**	**MIHI**	**Free**	**HW**	**Free HW**
$100K$	TR%	49.977	49.977	50.0	49.977	50.0
	FR%	0.0	0.0	0.0	0.0	0.0
	TA%	50.0	50.0	50.0	50.0	50.0
	FA%	0.023	0.023	0.0	0.023	0.0
$200K$	TR%	49.983	49.983	50.0	49.983	50.0
	FR%	0.0	0.0	0.0	0.0	0.0
	TA%	50.0	50.0	50.0	50.0	50.0
	FA%	0.017	0.017	0.0	0.017	0.0
$300K$	TR%	49.98	49.98	49.9993	49.98	49.9993
	FR%	0.0	0.0	0.0	0.0	0.0
	TA%	50.0	50.0	50.0	50.0	50.0
	FA%	0.02	0.02	0.0007	0.02	0.0007
$400K$	TR%	49.982	49.982	50.0	49.982	50.0
	FR%	0.0	0.0	0.0	0.0	0.0
	TA%	50.0	50.0	50.0	50.0	50.0
	FA%	0.018	0.018	0.0	0.018	0.0
$500K$	TR%	49.982	49.982	50.0	49.982	50.0
	FR%	0.0	0.0	0.0	0.0	0.0
	TA%	50.0	50.0	50.0	50.0	50.0
	FA%	0.018	0.018	0.0	0.018	0.0
$600K$	TR%	49.979	49.979	49.9995	49.979	49.9993
	FR%	0.0	0.0	0.0	0.0	0.0002
	TA%	50.0	50.0	50.0	50.0	49.9998
	FA%	0.021	0.021	0.0005	0.021	0.0007
$700K$	TR%	49.982	49.982	49.9997	49.982	49.9997
	FR%	0.0	0.0	0.0	0.0	0.0
	TA%	50.0	50.0	50.0	50.0	50.0
	FA%	0.018	0.018	0.0003	0.018	0.0003
$800K$	TR%	49.979	49.979	49.999875	49.979	49.99975
	FR%	0.0	0.0	0.0	0.0	0.000125
	TA%	50.0	50.0	50.0	50.0	49.999875
	FA%	0.021	0.021	0.000125	0.021	0.00025
$900K$	TR%	49.983	49.983	49.9998	49.983	49.9997
	FR%	0.0	0.0	0.0	0.0	0.0001
	TA%	50.0	50.0	50.0	50.0	49.9999
	FA%	0.017	0.017	0.0002	0.017	0.0003
$1M$	TR%	49.983	49.983	49.9996	49.983	49.9994
	FR%	0.0	0.0	0.0001	0.0	0.0003
	TA%	50.0	50.0	50.0	50.0	49.9998
	FA%	0.017	0.017	0.0003	0.017	0.0005

Consequently, the strategies are shown as viable solutions and do not impact the quality of the system.

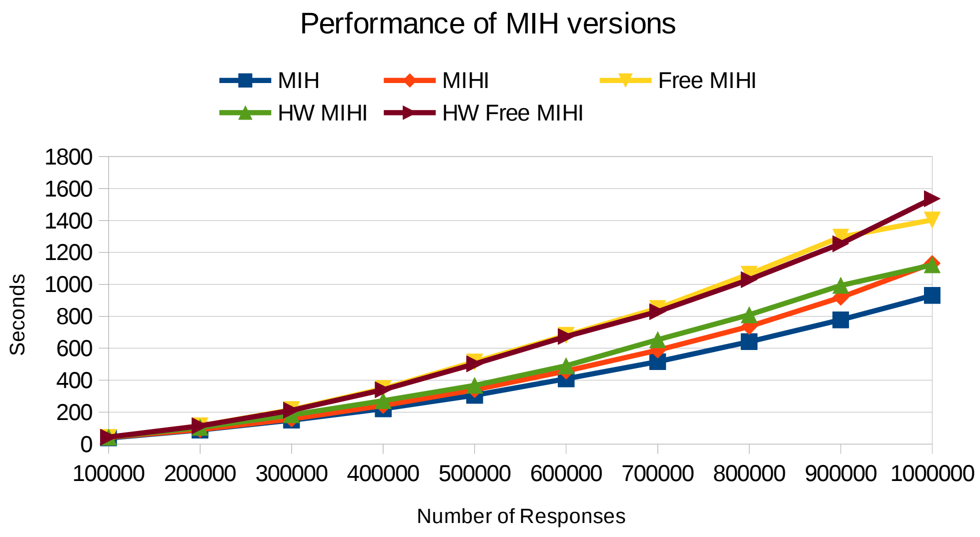


Figure 4.10: Performance of all MIH versions. Each line represents one version.

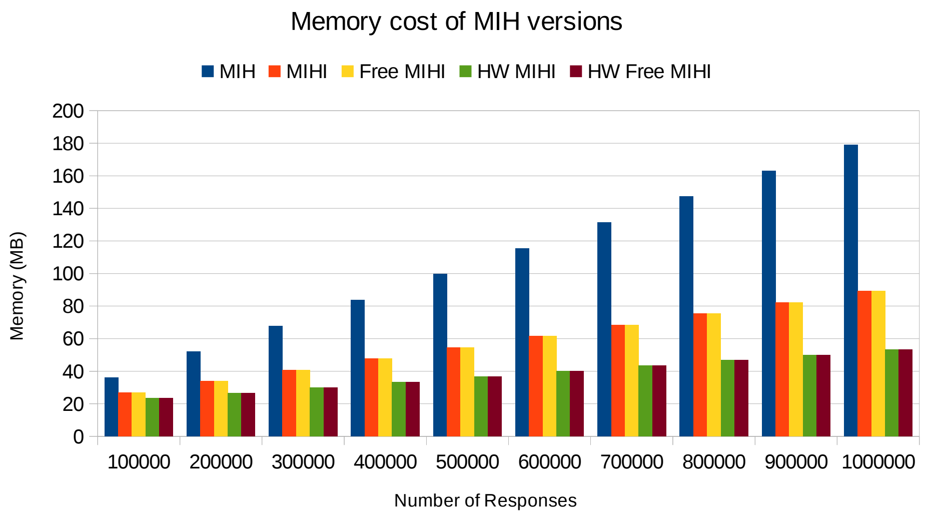


Figure 4.11: Memory consumption of all MIH versions. The memory axis is the total of the required space for MIH structure, response mapping table, database and hamming weight database.

The performance results are presented in Figure 4.10. The original MIH achieved the best results across the datasets. It was better than MIH with global index due to it took full advantage of cache locality in HD calculation during MIH search, accessing only 100 bits instead of 1000 bits from the database. The distance free computation versions obtained the worst results as MIH search cannot totally run in parallel. At some time, the hash tables have to be synchronized by index counter addition. The hamming weight versions exhibited competitive performance in comparison with the corresponding versions without the strategy.

Finally, the memory consumption results are presented in Figure 4.11. The total of memory space consists of the MIH structure, response mapping table, hamming weight database and the database, according to the requirements of each version. The hamming weight versions utilized fewer resources than other versions. By just using the global index within MIH, it was possible to considerably reduce the memory cost. Additionally, the MIH resources can still be reduced by adjusting the hash tables value length, since global index length was setup to 32 bits.

An interesting observation is that the two versions of MIH (Free MIHI and HW Free MIHI, HW MIHI and MIHI) with closely curves in Figure 4.10 do not have closely bars in Figure 4.11. This suggests a criteria for choosing which MIH implementation would be best suited for the available environment. In terms of memory consumption, Hamming Weight (HW) versions are the best options by achieving competitive performance compared to other versions. The strategies and global index optimization are practical alternatives which enable to save memory resources in trade-off of the performance loss.

4.5 Concluding Remarks

PUFs are expected to offer high uniqueness to ensure their unclonability property. To guarantee this property, online techniques to test manufactured PUFs in high volume manufacturing are required. This work analyzed an online testing methodology for Strong PUFs based on MIH and identified optimizations in order to reduce the memory consumption while keeping acceptable performance. Two MIH implementations were discussed and evaluated. Also, the distance free computation and hamming weight strategies were proposed to mitigate the memory cost. The results show that the memory cost can considerably be reduced at the expense of performance loss.

Chapter 5

Deeply-Nested Implicit Information Flow Tracking

Sensitive information leakage has been widely exploited by malicious attacks on personal computers like the recently reported Meltdown [93] and Spectre [94] attacks. As introduced in Section 2.3, information flow tracking is a useful mechanism in protecting software from malicious inputs. In *Dynamic Information Flow Tracking* (DIFT), a program identifies the sources of its untrusted inputs, and tracks the flow of such inputs through the course of its execution. DIFT may be used to prevent return address oriented attacks and malicious system calls [95].

DIFT consists of certain rules for tagging the initial data with a taint value and tracking it along the program execution path. During instruction execution, the taints associated with the source operands are propagated to the destination operand, tracking the information flow related to the data operations at runtime. At some point, an alarm is triggered when an inappropriate use of the tainted data is detected, for example, if it is copied to a data stream on the output channel. In previous works [52, 59, 96, 97], DIFT has been used as viable mechanism to detect attacks that leak secret information with explicit taint propagation.

Attackers can trick an explicit DIFT mechanism through insertion of implicit flows as discussed in [19], where the tracked sensitive data is propagated to untracked data by executing control flow instructions with no direct data transfers. This generates the error called *under-tainting* where data is not marked as tainted in cases that should be marked. In an execution flow of a conditional branch, the data value which influences the branch result may also affect data that is processed in an actual path. Hence, DIFT solutions should not only consider the *explicit propagation* to track the sensitive data, but also *implicit flow propagation* in order to defend against these advanced attacks.

To deal with implicit flow tracking, various software [63, 97] and hardware approaches have been proposed. Although they have shown to be effective, they exhibit some limitations regarding nested branches and loops which intensify the taint propagation originated from control-dependency flow. It causes *over-tainting*, that occurs when too many data is tainted degrading taint tracking precision.

This work provides the following contributions. First, a novel implicit flow tracking called NIFT - Nested Implicit Flow Tracking - is developed to support deeply-nested branches tracking by requiring few additional resources to extend explicit flow tracking approaches. NIFT efficiently solves the *under-tainting* problem and can be easily implemented in software DIFT solutions in spite of it was inspired by [70] which proposes a hardware-based implicit flow tracking. Second, a restricted rule to propagate taint when data is used in conditional branches is proposed. Only the immediate instructions without source operands receive the taint from conditional flow. This eliminates the *over-tainting* problem. Third, a formal verification is provided to prove the correctness of NIFT hardware specification. Lastly, the proposed mechanism is evaluated in terms of performance overhead, code size and implicit taint capabilities to deal with both *under-tainting* and *over-tainting* problems.

5.1 Nested Implicit Flow Tracking Implementation

In this Section, the restricted taint propagation for control flows and NIFT implementation are presented. Then, the code analysis and transformation algorithm to maintain the correct functionality of proposed scheme are described.

5.1.1 Taint Propagation to No-operand Instructions

Taint analysis might produce inaccurate results when not considering *implicit information flows*, resulting in *under-tainting* where data is not tainted although it is (indirectly) affected by a tainted source [63]. Moreover, implicit flows are critical in applications that require analyzing the information flow to detect a malware, because the adversary can inject complex implicit flows to evade detection.

Figure 5.1 shows an example of implicit flow. Intuitively, x and z are dependent on y even though there is no direct assignment among them, in other words, if y is tainted then x and z should also be tainted. From the adversary's point of view, the conditional branch allows to copy the value of the tainted y to x without using explicit propagation. Thus, considerable information can be copied to an untainted variable which is then used in the succeeding execution instead of the tainted data. Thereby, the new untainted data evades the taint check, allowing a malware writer to hide his attack so causing confidential information leakage. Within an *if statement*, all data operations will be affected by the variables used

```
z:= 2;
if (y = 1) then
{
    x := 1;
    z := 0;
}
else
    x := 0;
```

Figure 5.1: Example of code with implicit propagation flow.

in the conditional expression, in particular, the immediate assignment instructions that do not have source operands. Furthermore, there may be information flow even between a conditional branch and instructions which are not executed. For example, in Figure 5.1, z is not modified in *else* statement and in turn it will not be tainted. Although z remains unchanged, its value can leak information about the branch condition and, therefore, it must be tagged. The last case is beyond of the scope of this work, and one solution is presented in [70] that proposes a compensation technique to deal with untaken paths.

Conventional implicit flow tracking approaches propagate taint of data used in conditional branches to all data operations performed inside the conditional branch scope. In this way, an untainted data coming from outside the branch could be incorrectly marked as tainted inside the branch causing *over-tainting*, that is, data is marked as tainted when it should not be marked.

The attack described above, which copies tainted variable y to untainted variable x, is possible because x is computed by an immediate assignment instruction. From instruction level point of view, it is an immediate instruction without any source operands, but with a constant (immediate value); thus it has no data dependency. The following taint propagation by control dependencies is proposed based on this fact: when a taint propagation is originated from control dependencies, only no-operand instructions should receive taint information from branch result, while the remaining instructions should receive taint propagation by data dependency. This new rule reduces the amount of data tainted in branch scopes mitigating *over-tainting* problem.

5.1.2 Branch and Context Counter Scheme

As presented in Section 5.1.1, implicit flows occur when a program has control dependencies. Language-based static techniques [70] handle these implicit flows by including a program counter tag t_{PC} which informs if the current control path is affected by a tainted data. For every conditional branch, the taint of registers processed in the condition are propagated to t_{PC}. After that, all instructions executed in the taken path are tagged according to t_{PC} value, indicating they are affected by the branch result. Considering an implementation of this scheme for the example in Figure 5.1; if y is tainted then t_{PC} is set to 1 and, consequently, its propagation marks x and z as tainted. Thus, the implicit flow can be tracked by propagating t_{PC} value along the branch.

At a certain point in the program execution, the taken branch path is no longer affected by the conditional branch when it reaches the *immediate post-dominator* of the branch. For example, in Figure 5.2(a) the block $B5$ is the immediate post-dominator of block $B1$, which contains a conditional branch, because all paths from block $B1$ must pass through block $B5$ to exit the program. Therefore, the t_{PC} needs to be cleared when entering the immediate post-dominator block so that t_{PC} propagation does not overly tag the data outside the control path. By just clearing t_{PC} at the ending of branch does not work in case of multiple-nested branches. After exiting the inner branch, clearing t_{PC} disables the implicit propagation continuation in the outer branches. In [70], t_{PC} was replaced by an t_{PC} stack in order to remedy the multiple-nested branches situation. Before entering each conditional branch, the t_{PC} stack is pushed and, thereafter, taint propagation is done based on the t_{PC} value at the top of the stack. At the beginning of the immediate post-dominator block, the stack is popped recovering the t_{PC} value used in the previous branch.

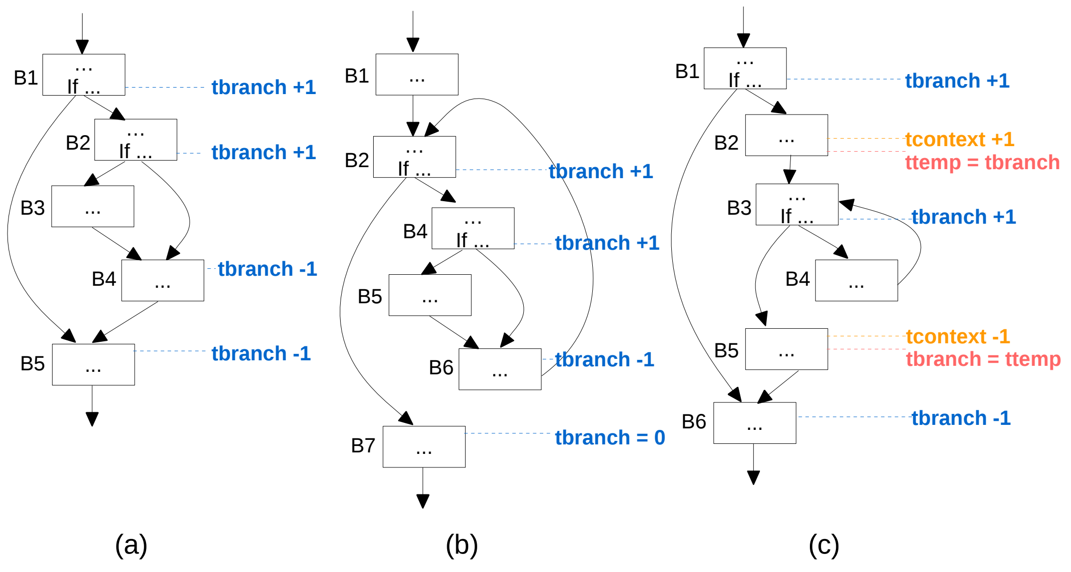


Figure 5.2: Examples of taint register operations.

In proposed approach, the program counter tag idea is extended by including a taint branch register t_{branch} and taint context register $t_{context}$. For every conditional branch, t_{branch} is incremented by 1 and, in each respective immediate post-dominator block, t_{branch} is decremented by 1 as depicted in Figure 5.2(a). During the taken branch execution, taint propagation follows our restricted rule defined in Section 5.1.1, where the variables that are assigned with an immediate value are tainted (tag is set to 1) if $t_{branch} > 0$. Only the immediate assignment instructions are considered as they have no dependency source making it is possible the transfer of a tracked value to an untainted variable as discussed in Section 5.1.1. Moreover, these immediate instructions are targeted by attackers that inject complex implicit flows into the program as discussed in Section 5.1.1. In addition, inner branches will always be tainted when the outer branch has been tainted. To sum up, t_{branch} will also be increased if t_{branch} is positive, even though the condition result is not tagged.

Figure 5.2(b) shows the loop scenario, where t_{branch} is increased every iteration. At the end of the loop, t_{branch} is cleared to avoid incorrect propagation outside the loop. In the cases of nested loops and a loop inside a conditional branch, t_{branch} has to recover its old value, obtained before starting the loop execution, at the exit of the loop. To remedy this, $t_{context}$ is increased by 1 and the temporary register t_{temp} stores the value of t_{branch} from the previous block of the nested loop header. At the end of the nested loop, $t_{context}$ is decreased and t_{branch} is restored from t_{temp}. The t_{temp} operation occurs only in the first level of the inner loop (where $t_{context}$ equals 0), since the inner loops inherit the taint from the outer loop even when the conditional expression is not tagged. The example in Figure 5.2(c) shows a loop executed in a conditional branch in which t_{temp} is handled similarly as in the nested loop case.

5.1.3 Taint Instructions

During compilation, specialized instructions will be inserted into the host code to manage the proposed taint registers. Generally, the final binary code is composed of the user program and the external libraries provided by the OS to allocate the required resources during the application execution. As NIFT can not guarantee the correct taint registers manipulation for the extra code inserted in the assembler phase, the taint status register t_{status} is included to flag when the implicit propagation is active, that is, when the instructions corresponding to the user program are executed. In this way, the conditional branch does not modify t_{branch} if it belongs to an external code outside the main function scope. The **initaint** instruction sets the active bit of t_{status} to 1 initializing the implicit propagation verification, while **haltaint** instruction halts this verification. Besides resetting t_{status}, **haltaint** clears the other taint registers.

At the beginning of the immediate post-dominator of each conditional branch, the **dtaint** instruction decrements t_{branch} by 1 and the **ztaint** instruction zeroes t_{branch} when there is only one loop. Figure 5.3(a) and (b) illustrate the modified

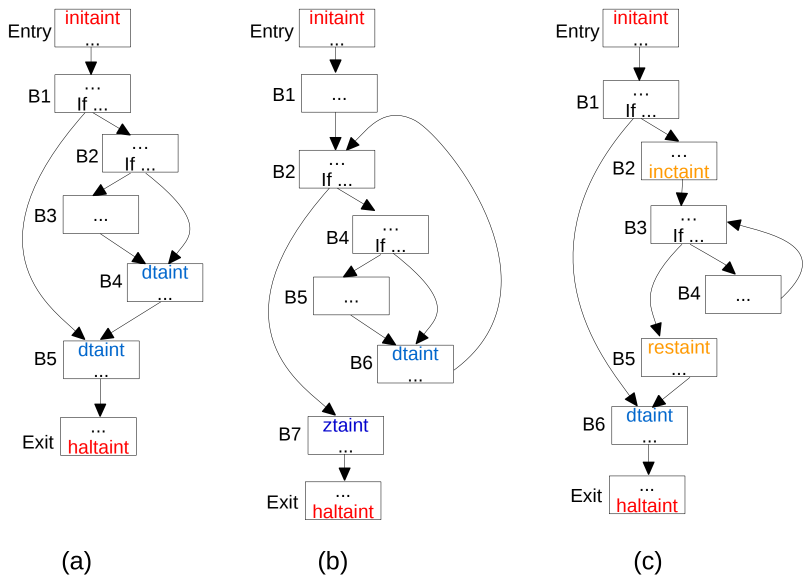


Figure 5.3: Examples of taint instructions.

code with **dtaint** and **ztaint**, respectively. Note that at entry block of the function, **initaint** enables the implicit propagation, that is disabled by **haltaint** when execution reaches the exit block. In the nested loop and loop in a branch cases, the **inctaint** instruction increments $t_{context}$ in each inner level and copies t_{branch} to t_{temp} before executing the first-level of the inner loop. Finally, the **restaint** instruction restores t_{branch} for the proper outer context propagation. The case of loop executed inside a branch is exemplified in Figure 5.3(c).

In Algorithm 1, the procedure for taint instruction placement into the original program at compile time is presented. **ztaint** and **dtaint** operations are inserted at the immediate post-dominator of the branch belonging or not to a single loop, respectively. While **inctaint** and **restaint** operations are inserted in case of nested loops or loop inside at least one branch. A point to emphasize is the **ztaint** placement, as it must be included in the *main* function. A function call may start anywhere during the loop iteration and, consequently, a single loop of a *callee* function might clear t_{branch} through a **ztaint** operation, disabling the implicit taint propagation upon returning to the *caller* function. As the implicit taint propagation is determined at runtime, all cases of loops are handled by **inctaint** and **restaint** operations when they are executed in the *non-main* functions. Furthermore, **initaint** and **haltaint** operations are only inserted into the *main* function. Real application codes usually use system calls as *exit(-1)* to abort the program execution from any part of the code, generating *unreachable* instructions from the compiler point of view. This situation is dealt at lines 17-19 in Algorithm 1, by inserting **haltaint** to clear the taint register before executing the unreachable instructions.

Another special case occurs when the conditional expression is composed of log-

Algorithm 1: Algorithm for inserting taint instructions.

```
   Input: Control flow graph of a function $F$
   Output: Control flow graph with taint instructions
1  foreach loop $l$ do
2      Find immediate post-dominator $p$ of exiting block(s) of $l$;
3      if $l$ is outer loop and $l$ is not inside an if block and $F$ is main function
        then
4          Insert ztaint at the start of $p$;
5      else
6          Insert inctaint at the end of the preheader of $l$;
7          Insert restaint at the start of $p$;
8      end
9  end
10 foreach non-loop conditional branch block $b$ do
11     Find $b$'s immediate post-dominator block $p$;
12     Insert dtaint at the start of $p$;
13     if $b$'s expression has logical "and" or "or" then
14         Insert dtaint before the conditional branch instruction of the block
            targeted from $b$;
15     end
16 end
17 foreach block $u$ containing a unreachable instruction do
18     Insert haltaint before the unreachable instruction;
19 end
20 if $F$ is main function then
21     Insert initaint at the start of the entry block;
22     Insert haltaint at the end of the last block;
23 end
```

ical *and* and *or*, as described in lines 13-15 of Algorithm 1. That expression is split into a chain of conditional blocks where each one increments t_{branch}. As it represents one condition, only the last conditional instruction should affect the branch and, therefore, for each conditional block targeted in this chaining, a **dtaint** instruction is inserted to correct the value to be used for implicit propagation.

5.2 Nested Implicit Flow Tracking Formal Verification

In order to verify the correctness of the proposed hardware mechanism introduced in Section 5.1, a formal verification tool called UPPAAL is utilized to design NIFT and prove its specification. This Section gives a brief introduction to the model checker UPPAAL, describes NIFT with UPPAAL model and discusses the verification results.

5.2.1 UPPAAL Model Checker

UPPAAL [98] is a state-of-the-art model checker based on theory of timed automata for real-time systems that consists of a model-checker engine and graphical user interface for modelling, simulation and verification.

Systems are modelled as a network of timed automata where each timed automata is a finite state machine with clocks. The nodes are *locations* that represent states indicating current values of system's variables, while the edges are *transitions* that change active location when a boolean condition on the variables and clocks is satisfied. Whenever a transition occurs, the variables can be modified and the clocks can be reseted. The timed automata is composed of concurrent processes that can use a synchronization channel to fire transitions at the same time. An edge labelled **name!** synchronizes with another timed automata having an edge labelled **name?**, where the condition that fires **name!** also activates **name?** so that both are executed as one transition. A location can be defined as *urgent* and *committed*, enabling the transition modifications be run without time delay. These specials locations allow to express atomic sequences in the system.

Other features are provided as C-like programming interface to ease the task description and property behavior definition. Verification of properties is performed through queries formulation used by UPPAAL verifier. Moreover, UPPAAL query language can express the following properties: *reachability*, *safety*, *liveness* and *deadlock*.

5.2.2 Modelling Nested Implicit Flow Tracking

NIFT is modelled as processor pipeline stage which communicates with decode stage of the pipeline. NIFT stage does not necessarily have to be subsequent to instruction decoder. As presented in Section 2.3.2, it might be implemented as an in-core design by extending pipeline of the main core or an off-core design by building it as a co-processor.

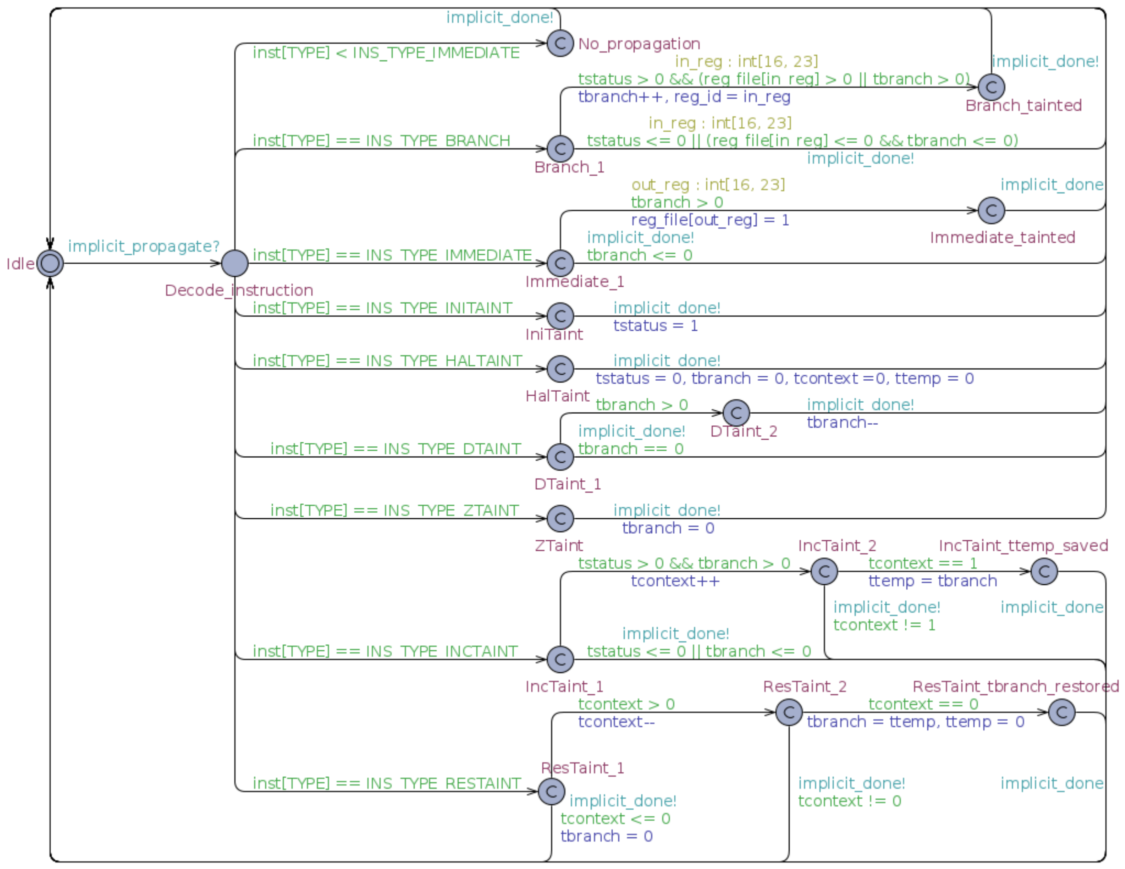


Figure 5.4: UPPAAL model for NIFT.

Figure 5.4 shows NIFT model designed with UPPAAL. The resources are defined for 32-bit MIPS processor that has 32 32-bit general-purpose registers and executes conditional branch in one instruction. The taint of MIPS registers is stored in *reg_-file* array and the taint registers are *tstatus*, *tbranch*, *tcontext* and *ttemp* variables. Initially, all variables and the program counter (*pc*) are zeroed and NIFT is in *Idle* location. Starter process fetches instruction and sends **implicit_propagate!** event to synchronize with NIFT **implicit_propagate?** transition, afterwards it waits to receive **implicit_done!** from NIFT to proceed to next instruction accumulating *pc* register. When NIFT fires **implicit_propagate?**, it changes to *Decode_instruction* location where the instruction opcode stored in *inst[TYPE]* is checked to determine the execution path. Taint instructions and conditional branches are executed according to the rules discussed in Section 5.1.3. To simple the design, one type is assigned to represent conditional branch instructions and immediate instructions which are no-operand such as “addi rt, $zero, immmediate” in MIPS. In branch and immediate execution, the input and output registers are randomly selected between the $16 to $23 saved registers employed by MIPS assembly convention. All execution paths are implemented as one atomic operation by using *committed* locations that are locations marked “C”. The other type of instructions are sent to *No_propagation* location where it only sends **implicit_done!** event to return to Starter process.

5.2.3 Correctness Verification

The correctness is verified as requirement specification through queries that allow to describe system properties along the all paths and states of the timed automata. The correctness of NIFT model, depicted in Figure 5.4, has successful verified by specifying the *reachability*, *safety*, *liveness* and *deadlock* properties using UPPAAL query language and running them in UPPAAL verifier. Starter process was initialized to fetch 100 instructions selected uniformly for each instruction type, sending them to NIFT automaton. The queries are formulated with the following specifications:

- *Reachability* - this property guarantees that some state is reachable from initial state. This property is expressed by UPPAAL query language as following:
  - $E <> p$: there exists a path where p eventually occurs, where E represents some paths, $<>$ represents some states and p is the property. An example is shown in Figure 5.5.

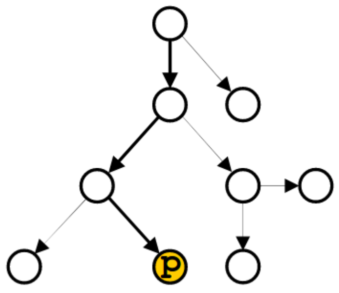
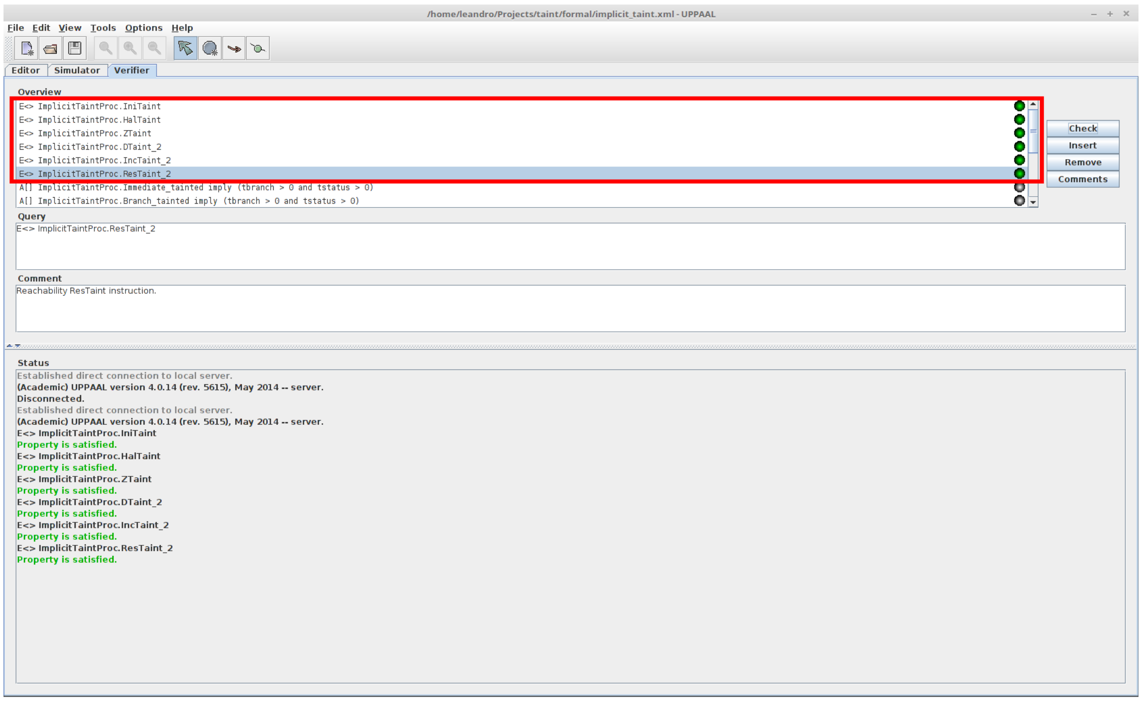


Figure 5.5: Example of reachability property on UPPAAL.

  The queries for reachability property are presented in Figure 5.6. All execution path locations of NIFT have to be reachable from initial *Idle* location by some path. In this way, it can properly execute branch, immediate and taint instructions.

- *Safety* - it guarantees that a state is only active when all required conditions are true. There are two forms to express safety using UPPAAL query language:



Overview

```
E<> ImplicitTaintProc.IniTaint
E<> ImplicitTaintProc.HalTaint
E<> ImplicitTaintProc.ZTaint
E<> ImplicitTaintProc.DTaint_2
E<> ImplicitTaintProc.IncTaint_2
E<> ImplicitTaintProc.ResTaint_2
A[] ImplicitTaintProc.Immediate_tainted imply (tbranch > 0 and tstatus > 0)
A[] ImplicitTaintProc.Branch_tainted imply (tbranch > 0 and tstatus > 0)
```

Query

```
E<> ImplicitTaintProc.ResTaint_2
```

Comment

Reachability ResTaint instruction.

Status

```
Established direct connection to local server.
(Academic) UPPAAL version 4.0.14 (rev. 5615), May 2014 -- server.
Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.0.14 (rev. 5615), May 2014 -- server.
E<> ImplicitTaintProc.IniTaint
Property is satisfied.
E<> ImplicitTaintProc.HalTaint
Property is satisfied.
E<> ImplicitTaintProc.ZTaint
Property is satisfied.
E<> ImplicitTaintProc.DTaint_2
Property is satisfied.
E<> ImplicitTaintProc.IncTaint_2
Property is satisfied.
E<> ImplicitTaintProc.ResTaint_2
Property is satisfied.
```

Figure 5.6: Reachability property of NIFT on UPPAAL verifier. There are six queries to verify if the state of the taint instructions is reachable.

- $E[]p$: there is an execution path in which p occurs for all the states of the path, where E represents some paths, [] represents all states and p is the property. It is exemplified in Figure 5.7(a).
- $A[]p$: For each (all) execution path, p occurs for all the states of the path, where A represents all paths, [] represents all states and p is the property. It is exemplified in Figure 5.7(b).

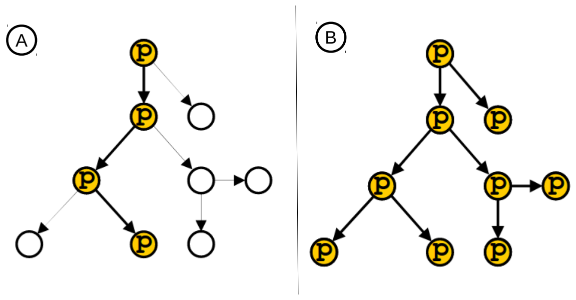
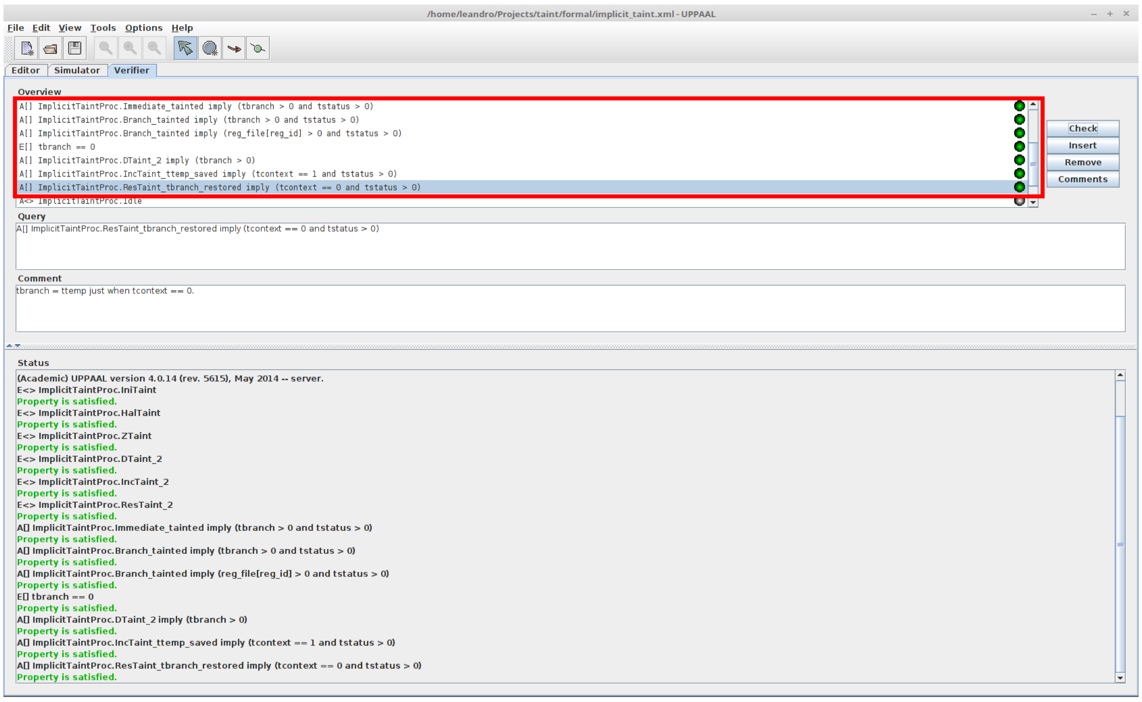


Figure 5.7: Example of safety property on UPPAAL.

In NIFT, safety is defined for the execution path of the taint instructions, where they must just be activated when the taint registers' values are complied with criteria established in Section 5.1.3. The queries for safety property are presented in Figure 5.8.

- *Liveness* - it determines that some state is eventually activated in certain


/home/leandro/Projects/taint/formal/implicit_taint.xml - UPPAAL
File Edit View Tools Options Help
Editor Simulator Verifier
Check
Insert
Remove
Comments

Overview

```
A[] ImplicitTaintProc.Immediate_tainted imply (tbranch > 0 and tstatus > 0)
A[] ImplicitTaintProc.Branch_tainted imply (tbranch > 0 and tstatus > 0)
A[] ImplicitTaintProc.Branch_tainted imply (reg_file[reg_id] > 0 and tstatus > 0)
E[] tbranch == 0
A[] ImplicitTaintProc.DTaint_2 imply (tbranch > 0)
A[] ImplicitTaintProc.IncTaint_ttemp_saved imply (tcontext == 1 and tstatus > 0)
A[] ImplicitTaintProc.ResTaint_tbranch_restored imply (tcontext == 0 and tstatus > 0)
A<> ImplicitTaintProc.Idle
```

Query

```
A[] ImplicitTaintProc.ResTaint_tbranch_restored imply (tcontext == 0 and tstatus > 0)
```

Comment

```
tbranch = ttemp just when tcontext == 0.
```

Status

```
(Academic) UPPAAL version 4.0.14 (rev. 5615), May 2014 -- server.
E<> ImplicitTaintProc.IniTaint
Property is satisfied.
E<> ImplicitTaintProc.HalTaint
Property is satisfied.
E<> ImplicitTaintProc.ZTaint
Property is satisfied.
E<> ImplicitTaintProc.DTaint_2
Property is satisfied.
E<> ImplicitTaintProc.IncTaint_2
Property is satisfied.
E<> ImplicitTaintProc.ResTaint_2
Property is satisfied.
A[] ImplicitTaintProc.Immediate_tainted imply (tbranch > 0 and tstatus > 0)
Property is satisfied.
A[] ImplicitTaintProc.Branch_tainted imply (tbranch > 0 and tstatus > 0)
Property is satisfied.
A[] ImplicitTaintProc.Branch_tainted imply (reg_file[reg_id] > 0 and tstatus > 0)
Property is satisfied.
E[] tbranch == 0
Property is satisfied.
A[] ImplicitTaintProc.DTaint_2 imply (tbranch > 0)
Property is satisfied.
A[] ImplicitTaintProc.IncTaint_ttemp_saved imply (tcontext == 1 and tstatus > 0)
Property is satisfied.
A[] ImplicitTaintProc.ResTaint_tbranch_restored imply (tcontext == 0 and tstatus > 0)
Property is satisfied.
```

Figure 5.8: Safety property of NIFT on UPPAAL verifier. There are seven queries to verify the correct management of the taint registers.

conditions. It is expressed as follows:

- $A <> p$: for each (all) execution path, p occurs for at least one state of the path, where A represents all paths, $<>$ represents some states and p is the property.

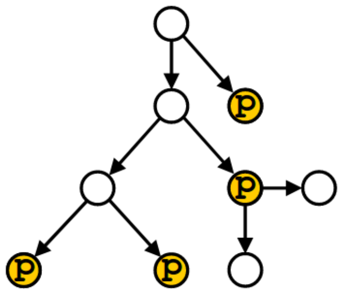
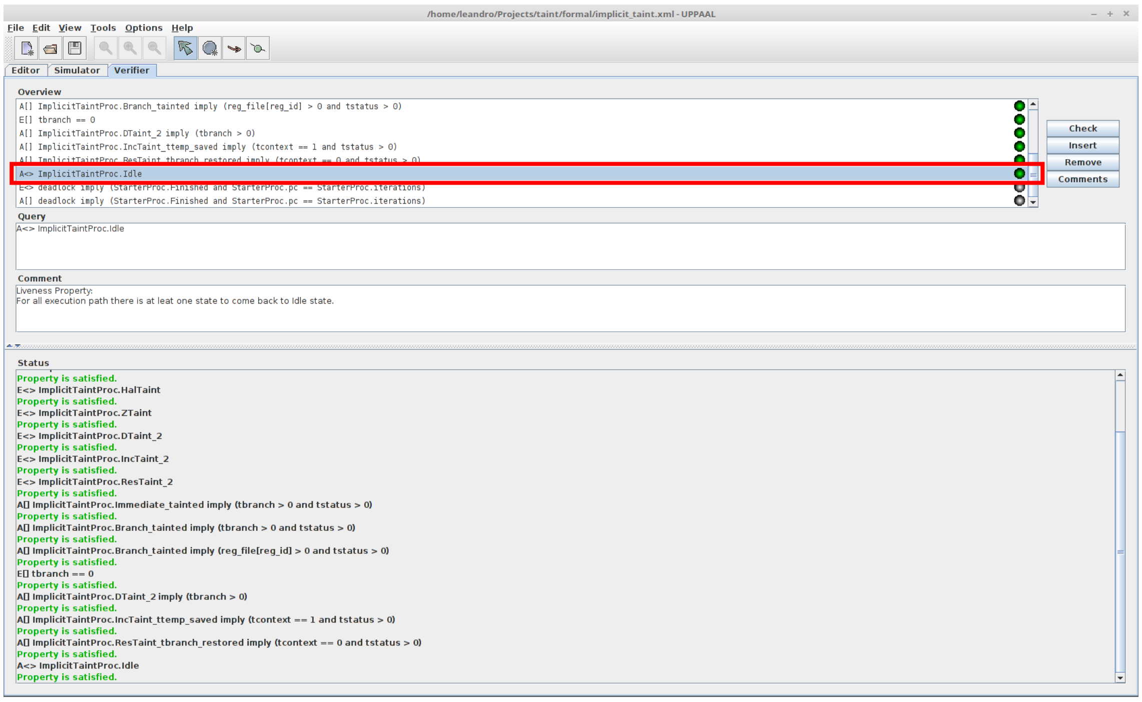


Figure 5.9: Example of liveness property on UPPAAL.

As presented in Figure 5.10, NIFT returns to *Idle* location after passed some time whenever it receives **implicit_propagate?** event from Starter process.

- *Deadlock* - NIFT is deadlock-free during instruction execution. It can be expressed by two forms:

  - $E <> deadlock$: exists deadlock, where E represents some paths and $<>$ represents some states;


/home/leandro/Projects/taint/formal/implicit_taint.xml - UPPAAL
File Edit View Tools Options Help
Editor Simulator Verifier
Overview
Check
Insert
Remove
Comments
Query
Comment
Status

```
A[] ImplicitTaintProc.Branch_tainted imply (reg_file[reg_id] > 0 and tstatus > 0)
E[] tbranch == 0
A[] ImplicitTaintProc.DTaint_2 imply (tbranch > 0)
A[] ImplicitTaintProc.IncTaint_ttemp_saved imply (tcontext == 1 and tstatus > 0)
A[] ImplicitTaintProc.ResTaint_tbranch_restored imply (tcontext == 0 and tstatus > 0)
A<> ImplicitTaintProc.Idle
E<> deadlock imply (StarterProc.Finished and StarterProc.pc == StarterProc.iterations)
A[] deadlock imply (StarterProc.Finished and StarterProc.pc == StarterProc.iterations)
```

Query:

```
A<> ImplicitTaintProc.Idle
```

Comment:

Liveness Property:
For all execution path there is at leat one state to come back to Idle state.

Status:

```
Property is satisfied.
E<> ImplicitTaintProc.HalTaint
Property is satisfied.
E<> ImplicitTaintProc.ZTaint
Property is satisfied.
E<> ImplicitTaintProc.DTaint_2
Property is satisfied.
E<> ImplicitTaintProc.IncTaint_2
Property is satisfied.
E<> ImplicitTaintProc.ResTaint_2
Property is satisfied.
A[] ImplicitTaintProc.Immediate_tainted imply (tbranch > 0 and tstatus > 0)
Property is satisfied.
A[] ImplicitTaintProc.Branch_tainted imply (tbranch > 0 and tstatus > 0)
Property is satisfied.
A[] ImplicitTaintProc.Branch_tainted imply (reg_file[reg_id] > 0 and tstatus > 0)
Property is satisfied.
E[] tbranch == 0
Property is satisfied.
A[] ImplicitTaintProc.DTaint_2 imply (tbranch > 0)
Property is satisfied.
A[] ImplicitTaintProc.IncTaint_ttemp_saved imply (tcontext == 1 and tstatus > 0)
Property is satisfied.
A[] ImplicitTaintProc.ResTaint_tbranch_restored imply (tcontext == 0 and tstatus > 0)
Property is satisfied.
A<> ImplicitTaintProc.Idle
Property is satisfied.
```

Figure 5.10: Liveness property of NIFT on UPPAAL verifier. For all instruction execution paths, the automaton reaches the Idle state.

  - *A*[] *not deadlock*: there is no deadlock, where *A* represents all paths and [] represents all states;

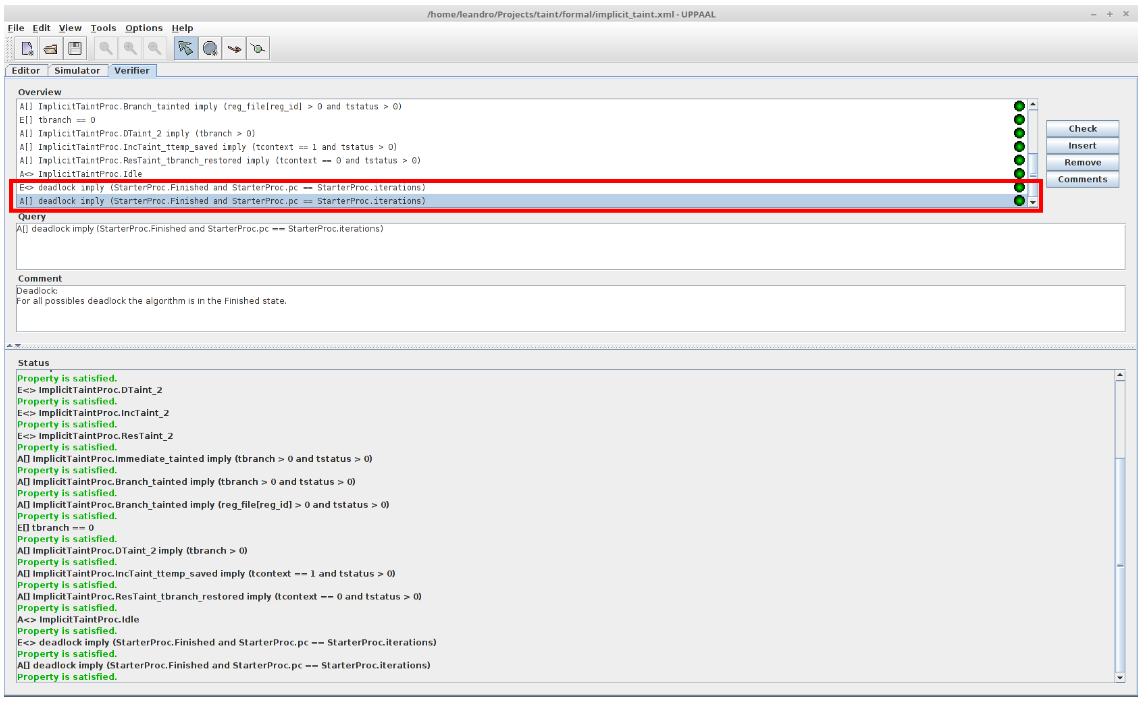
The queries for deadlock property are shown in Figure 5.11, which check if the program counter is updated with all executed instructions.

5.3 Experiment Results

This Section evaluates the overhead of the proposed deeply-nested implicit propagation flow technique, discussed in Section 5.1, in terms of performance and code size. Then, its ability to solve the *under-tainting* and *over-tainting* problem is analyzed.

5.3.1 Experimental Setup

To test the proposed NIFT, it was implemented in a modified version of the *libdft* tool [61], that is a software implementation for Dynamic Taint Analysis based on Intel's Pin dynamic binary instrumentation framework [72]. *libdft* only implements the explicit taint propagation on x86 architecture. The modified version supports a 64-bit architecture extended with 32-bit taint registers and the required logic to handle the implicit propagation. The code analysis and transformation described in Algorithm 1 are implemented in the LLVM compiler framework [99]. Taint instructions are inserted as special instructions with *opcodes* that are unused by the


/home/leandro/Projects/taint/formal/implicit_taint.xml - UPPAAL
File Edit View Tools Options Help
Editor Simulator Verifier
Check
Insert
Remove
Comments

Overview

```
A[] ImplicitTaintProc.Branch_tainted imply (reg_file[reg_id] > 0 and tstatus > 0)
E[] tbranch == 0
A[] ImplicitTaintProc.DTaint_2 imply (tbranch > 0)
A[] ImplicitTaintProc.IncTaint_ttemp_saved imply (tcontext == 1 and tstatus > 0)
A[] ImplicitTaintProc.ResTaint_tbranch_restored imply (tcontext == 0 and tstatus > 0)
A<> ImplicitTaintProc.Idle
E<> deadlock imply (StarterProc.Finished and StarterProc.pc == StarterProc.iterations)
A[] deadlock imply (StarterProc.Finished and StarterProc.pc == StarterProc.iterations)
```

Query

```
A[] deadlock imply (StarterProc.Finished and StarterProc.pc == StarterProc.iterations)
```

Comment

Deadlock:
For all possibles deadlock the algorithm is in the Finished state.

Status

```
Property is satisfied.
E<> ImplicitTaintProc.DTaint_2
Property is satisfied.
E<> ImplicitTaintProc.IncTaint_2
Property is satisfied.
E<> ImplicitTaintProc.ResTaint_2
Property is satisfied.
A[] ImplicitTaintProc.Immediate_tainted imply (tbranch > 0 and tstatus > 0)
Property is satisfied.
A[] ImplicitTaintProc.Branch_tainted imply (tbranch > 0 and tstatus > 0)
Property is satisfied.
A[] ImplicitTaintProc.Branch_tainted imply (reg_file[reg_id] > 0 and tstatus > 0)
Property is satisfied.
E[] tbranch == 0
Property is satisfied.
A[] ImplicitTaintProc.DTaint_2 imply (tbranch > 0)
Property is satisfied.
A[] ImplicitTaintProc.IncTaint_ttemp_saved imply (tcontext == 1 and tstatus > 0)
Property is satisfied.
A[] ImplicitTaintProc.ResTaint_tbranch_restored imply (tcontext == 0 and tstatus > 0)
Property is satisfied.
A<> ImplicitTaintProc.Idle
Property is satisfied.
E<> deadlock imply (StarterProc.Finished and StarterProc.pc == StarterProc.iterations)
Property is satisfied.
A[] deadlock imply (StarterProc.Finished and StarterProc.pc == StarterProc.iterations)
Property is satisfied.
```

Figure 5.11: Deadlock property of NIFT on UPPAAL verifier.

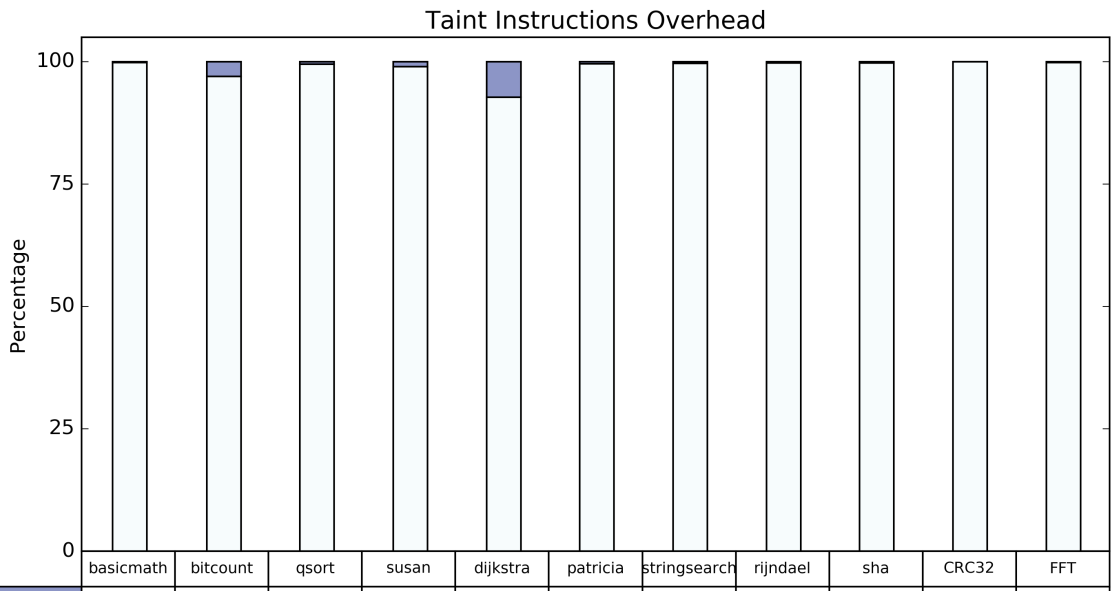
host processor architecture. Thus, the host processor executes them as *nop* operations. For the experiments, 11 applications are chosen from the *mibench* benchmark suite [100] and the taint sources are defined according to the standard libdft-DTA implementation which marks the external files as initial tainted data.

5.3.2 Performance Overhead

To measure the performance overhead, a Pin tool was implemented to count all the instructions dynamically executed at runtime. For each application, it receives the taint-binary version including the extra taint instructions. The small and large inputs were run for each application and the mean of the obtained results was calculated for the different inputs.

The results are shown in Figure 5.12. Overall, the total number of the executed taint instructions constitutes a small portion of all executed instructions, resulting in $< 3\%$ overhead for most applications and, in the worst case, approximately 7.3% overhead for *dijkstra*. Figure 5.13 shows the detailed overhead contribution of each taint instruction. The instruction with the highest overhead is **dtaint** that is executed at the end of each conditional branch and, consequently, is executed in each loop iteration. Nested loop cases increase the **dtaint** overhead and also increase the **inctaint** and **restaint** overhead.

Figure 5.14 shows the number of times each taint instruction is executed for *dijkstra* that has the highest overhead. The instructions are in execution order starting



	basicmath	bitcount	qsort	susan	dijkstra	patricia	stringsearch	rijndael	sha	CRC32	FFT
Total Taint	0.160316	2.972524	0.468726	1.029236	7.250017	0.436145	0.348043	0.230543	0.274798	5e-06	0.196599
Non-Taint	99.839684	97.027476	99.531274	98.970764	92.749983	99.563855	99.651957	99.769457	99.725202	99.999995	99.803401

Figure 5.12: Performance overhead of the NIFT(%).

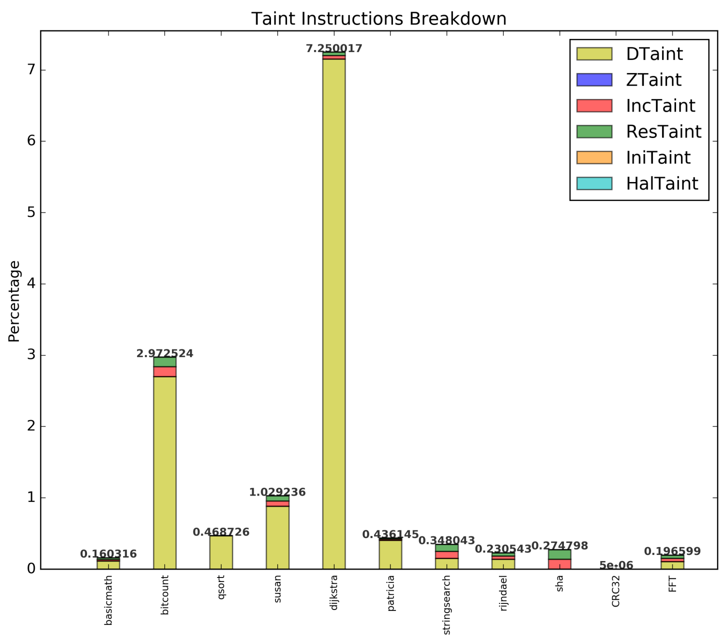
from bottom. By observing the order of **inctaint**, **restaint** and **ztaint** execution, it executes 4 nested loops where the same 3 **dtaint** operations are executed almost 10 million times, which explains its high overhead.

5.3.3 Code Size

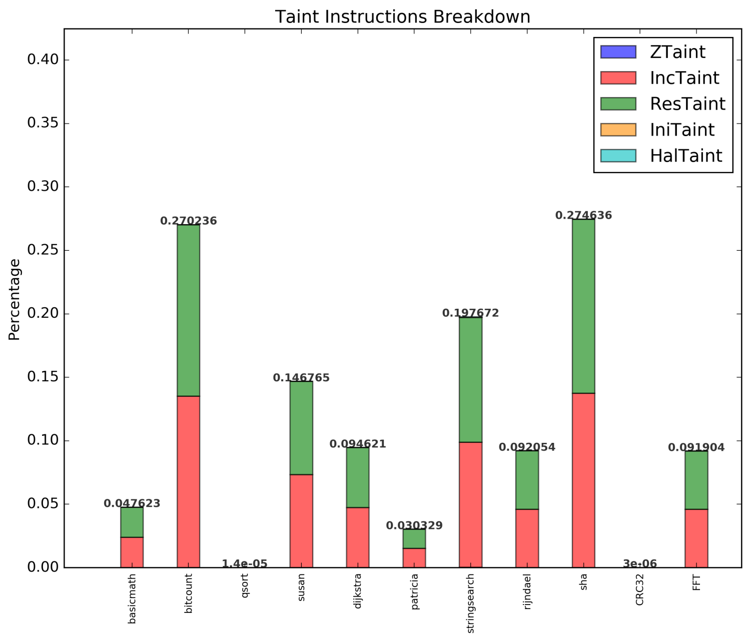
By including additional taint instructions in the binary code, it increases the required memory size to load the instructions before starting the program execution. To evaluate the overhead in term of memory size, a Pin tool was implemented to count statically the amount of bytes consumed for each instruction. The larger version of each application provided by the benchmark was selected. The total number of instructions and the memory consumption are tabulated in Table 5.1. The results show that the taint instructions add a small memory overhead constituting less than 0.3% of the whole program size. This low consumption enables NIFT mechanism to be performed at low resource systems like Android devices.

5.3.4 Tainting Capabilities

Since explicit DIFT approach exhibits the *under-tainting* problem, the extra data marked as tainted by the implicit information flow indicates the degree of resolving the *under-tainting* problem. On the other hand, a conservative control-dependency taint propagation strategy can lead to *over-tainting* problem as described in Section 2.3. To evaluate the accuracy of NIFT, a program that calculates 1002 digits of π



(a) Overhead of all taint instructions.



(b) Detailed overhead of taint instructions excluding Dtaint.

Figure 5.13: The overhead of the individual taint instructions.

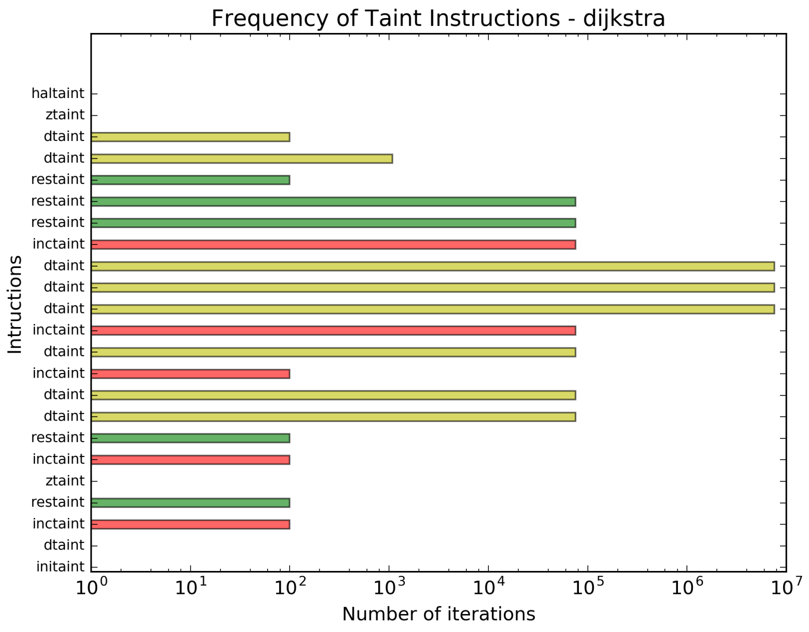
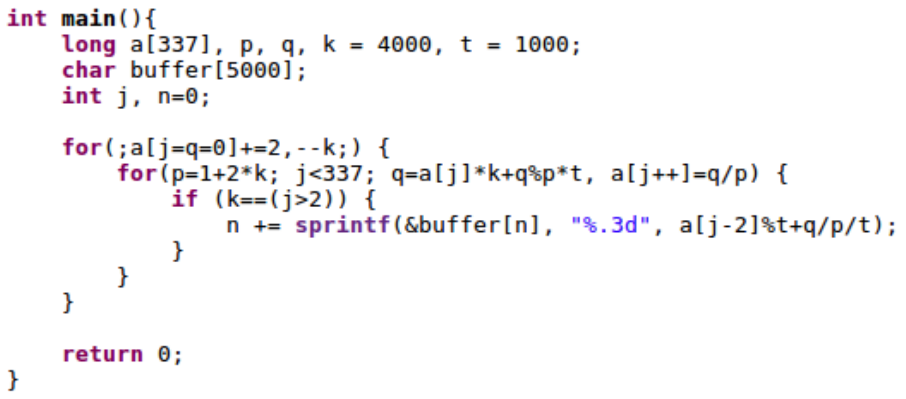


Figure 5.14: Number of iterations for the **dtaint** instructions.

Table 5.1: Number of instructions and code size (bytes).

Application	**# Total Insts**	**Bytes of Total Insts**	**# Taint Insts**	**Bytes of Taint Insts**
basicmath	189,460	817,815	27	108
bitcount	156,519	658,783	33	132
qsort	167,869	710,301	11	44
susan	176,050	738,037	383	1,532
dijkstra	155,519	655,310	24	96
patricia	156,544	659,261	90	360
stringsearch	156,084	657,609	83	332
rijndael	160,141	673,647	87	348
sha	138,546	559,957	26	104
CRC32	155,669	655,566	10	40
FFT	172,347	734,546	44	176


```
int main(){
    long a[337], p, q, k = 4000, t = 1000;
    char buffer[5000];
    int j, n=0;

    for(;a[j=q=0]+=2,--k;) {
        for(p=1+2*k; j<337; q=a[j]*k+q%p*t, a[j++]=q/p) {
            if (k==(j>2)) {
                n += sprintf(&buffer[n], "%.3d", a[j-2]%t+q/p/t);
            }
        }
    }

    return 0;
}
```

Figure 5.15: Program to compute digits of π.

was implemented based on [70]. The program is depicted in Figure 5.15. It uses *sprintf* function copied from *dietlibc* which converts a value to ASCII representation. This program is adequate to analyze the implicit flow tracking technique, because it is composed to implicit and explicit flows and has 7 nested branches (3 loops and 4 if): 3 in the *main* function and 4 in the *sprintf* function. NIFT is compared with both *libdft* and DYTAN. Since *libdft* implements just explicit propagation, its result represents the lower bound of *under-tainting*. On the other hand, DYTAN causes *over-tainting* indicating the upper bound. DYTAN's taint propagation was implemented on Pin. The *sprintf* function was slightly modified by inserting implicit flows in the part that transforms decimal digits to ASCII form. Initially, the memory location reserved for array a is tainted. At the end of *main* function execution, *buffer* should be tainted by ASCII translation.

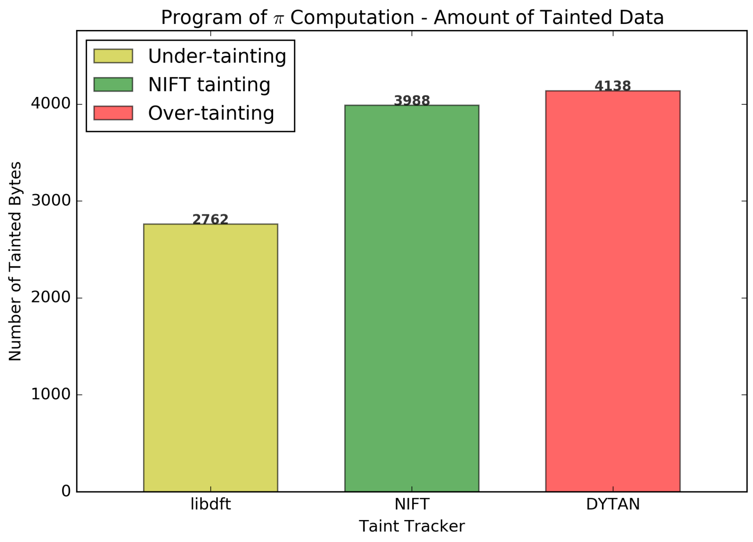
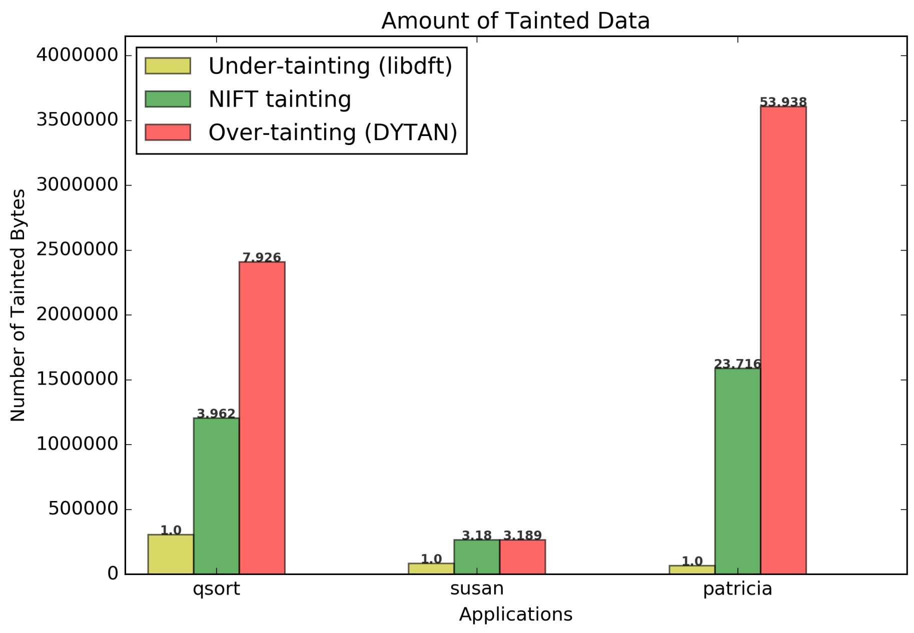


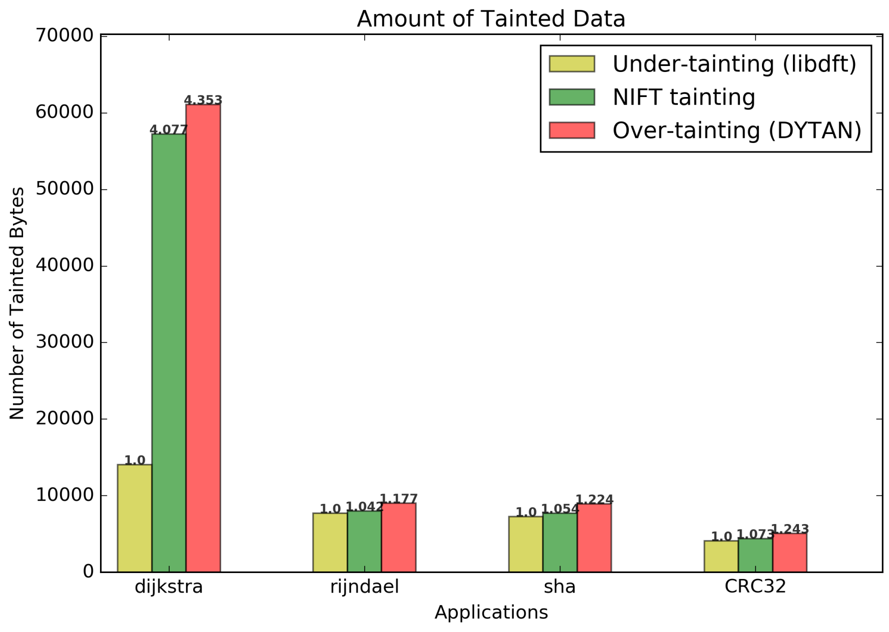
Figure 5.16: Comparison of tainted bytes in π computation program.

The results are shown in Figure 5.16. The tainted bytes were obtained at the end of *main* function. Total of $2,696$ ($337 \times 8bytes$) bytes are related to array a and $1,003$ (the one extra byte is tainted by *sprintf* which assigns 0 in the last location *buffer[1002]*) bytes are related to *buffer*. libdft does not taint *buffer* since implicit flow was added in *sprintf*. Both NIFT and DYTAN marked correctly $1,003$ *buffer* locations as tainted, whereas NIFT was accurate without overly marking data.

Figure 5.17 shows the comparison of tainted bytes using NIFT, DYTAN and *libdft* for 7 mibench applications which mark input file as tainted. For all applications, NIFT achieves better taint propagation results, where in the best case it reduces about 30.3 times the amount of tainted bytes compared to DYTAN for *patricia* application. It was also accurate in security applications *sha* and *rijndael*. The analysis of the results shows that the proposed NIFT effectively tracks the implicit



(a) Tainted bytes of qsort, susan and patricia applications.



(b) Tainted bytes of dijstra, rijndael, sha and CRC32 applications.

Figure 5.17: Comparison of tainted bytes for mibench applications. Each bar has the ratio of bytes considering under-tainting result as baseline.

information flow even in deeply-nested scenario without generating *over-tainting*.

5.4 Concluding Remarks

Since the main goal is to add implicit taint tracking to existing explicit propagation solutions, the required resources for hardware implementation are analyzed. Primarily, the resources are the four 32-bit taint registers, the t_{branch} counter, the $t_{context}$ counter and the nested implicit flow control unit. As $t_{context}$ increments for each nested loop level, its counter may be implemented using simple shift operations by executing one left shift and right shift to increase and decrease its value, respectively. The nested implicit flow control unit is responsible for detecting the conditional branch instructions, immediate assignment instructions and taint instructions, forwarding the execution to the logic circuit for each type of instruction as modelled in Section 5.2.2. The remaining resources consist of the control logic to handle the taint registers and the taint propagation from t_{branch} to the general-purpose registers. Those resources are extended at low cost in the hardware designs presented in Section 2.3.2. Future work will focus on exploring mechanism to alarm taint analysis using a taint counter approach for specific security applications.

NIFT can be easily implemented to extend explicit DIFT hardware and software solutions. It provides an accurate nested implicit flow tracking ability with low cost through management of four taint registers instead of using management of extra stack resources as proposed in [70].

To complete the proposed hardware additions, a simple method is described to start the taint data analysis in order to protect confidential data and avoid its leakage. A typical example is to protect the memory region where the AES cipher's private key is stored. Initially, the AES memory region should be tainted, for example, by running an OS system call that allocates a memory space tagging it. During the taint propagation, a taint counter $t_{counter}$ increments whenever a new output operand is marked as tainted. A threshold is determined to limit $t_{counter}$ and a supervisor process checks the $t_{counter}$ value. When $t_{counter}$ value exceeds the given threshold, the taint analysis is invoked to detect suspicious activities. By detecting successfully a malicious access, the process might be aborted. Otherwise, $t_{counter}$ is zeroed and the execution goes on.

Conclusion

Dynamic Information Flow Tracking is an appealing approach to detect malicious operations at runtime. However, implicit propagation flow results in *under-tainting* which can be exploited by the adversary to evade tracking. This work proposes NIFT

– a Nested Implicit Flow Tracking – which can be easily added to the conventional information flow tracking architecture. The new approach includes taint registers to tag the data affected by control flow and relies on the compiler to handle correctly the scope of the implicit propagation tracking. To mitigate the *over-tainting* problem, a new taint rule is applied to propagate it to no-operand instructions like immediate assignments instructions in control-dependency. Furthermore, a formal verification of NIFT hardware design is developed to prove its correctness. The experimental results demonstrate a low overhead in terms of performance and memory consumption. NIFT was effective in catching implicit information flows and our taint propagation improve the accuracy to taint data in deeply-nested cases without generating *over-tainting*. The proposed portable implicit information tracking is a general approach that can be easily implemented in hardware and software explicit information flow mechanisms with low resources overhead through management of four taint registers.

Chapter 6

Memory Efficient WiSARD using Approximate Membership Query

Weightless Neural Network (WNN) is an abstract model of biological neuron where each neuron is defined as Random Access Memory (RAM) node [47]. As it can be implemented using existing memory resources in devices and the neurons are represented in a binary format, this model offers attractive practical solutions to solve pattern recognition and artificial consciousness applications achieving competitive performance. WiSARD (Wilkie, Stoneham and Aleksander's Recognition Device) is the simplest WNN model inspired by the n-tuple classifier [49] that provides efficient implementation using standard RAM memory technology, enabling to deploy learning capabilities into real-time and embedded systems as introduced in Section 2.2.

In certain applications, the straightforward WiSARD implementation requires a considerable amount of memory resources to achieve good learning features. For example, a 1024×1024 binary input with total size of $1,048,576$ bits can be mapped into 64-bit tuples to $16,384$ RAMs ($64 \times 16,384 = 1,048,576$). In this configuration, each RAM would consume 2^{64} locations which is impracticable to be implemented in actual technology. To deal with this constraint, the RAMs are commonly implemented using dictionary/hash table structures where the tuples values are stored as key-value pair at each position, with the key representing the memory address and, the value, the tuple [101].

To address this problem, this Chapter proposes novel WiSARD models based on Approximate Membership Query (AMQ) structures which are probabilistic data structures to represent a set using less space than popular dictionary implementations [102]. Most relevant AMQ structures: Bloom filter, Cuckoo filter and Quotient filter, are explored to extend the original WiSARD model. Finally the proposed models are compared to the standard WiSARD and WiSARD implemented with hash table.

6.1 Approximate Membership Query Structures

Approximate Membership Query data structures maintain a probabilistic representation of a set S by providing lookup and insert operations over the elements on it. They offer a space efficiency through a trade-off: the membership query operation is approximate. For an element $e \in S$, $LOOKUP(e)$ responds “present”, while for $e \notin S$, $LOOKUP(e)$ can also respond “present” with a probability at most ϵ which indicates the false-positive rate. In other words, sometimes the membership query will respond that an element was stored while actually it was not inserted. AMQs have been adopted in various kind of applications such as networks, storage systems, databases, computational biology and other domains. Moreover, there are AMQ structures that extend the supported number of operations by including counting and delete operations.

This Section provides a brief overview of the major AMQ data structures, following the description of Bloom Filter, Cuckoo Filter and Quotient Filter.

6.1.1 Bloom Filter

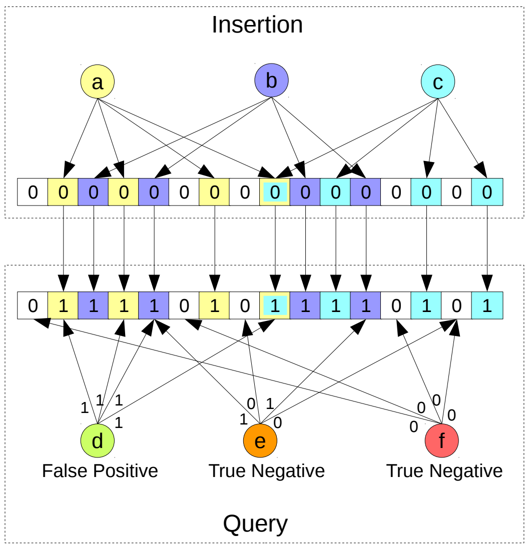


Figure 6.1: Bloom filter operations example with 16-bit array and 4 hash functions.

Bloom filter is the most well-known space-efficient data structure for AMQ which tests whether an element belongs to a given set or not with a certain false positive probability [103]. A Bloom filter consists of an m-bit array and k independent hash functions that map an element into k bit array positions.

Figure 6.1 exemplifies the insertion and query operations supported by the standard Bloom filter. Initially, all bit array positions are zeroed. In the insertion operation, an element is mapped into k positions of the bit array using the k hash functions and the corresponding k bits are set to 1. In the example, a, b and c are inserted using 4 hash functions. The query operation looks up the k positions mapped from the input element, informing it as either a member of set, with a false positive rate if all values are $1's$, or a non-member when any value is 0. In Figure 6.1, d is a false positive element since it was suggested as member of set (only a, b and c were inserted), while e and f do not belong to the set. Note that a Bloom filter correctly reports a true negative whenever an element is not a member.

The probability of false positive p is affected by the parameters m, n and k, corresponding to bit array size, number of elements to store and number of hash functions, respectively [104].

$$m = -n \times \frac{\ln(p)}{\ln(2)^2} \tag{6.1}$$

$$k = m \times \frac{\ln(2)}{n} \tag{6.2}$$

Given the probability p and capacity n, it is possible to determine the ideal parameters m and k. The number of bits m is calculated by the Formula (6.1) [105], while the number of hash functions k is obtained by the Formula (6.2) [104].

6.1.2 Cuckoo Hashing

Cuckoo hash table is an efficient dictionary data structure which utilizes the cuckoo hashing technique to resolve the hash collisions [106]. A basic cuckoo hash table consists of an array of buckets where each item can be only stored in two candidate buckets specified by hash functions $h_1(x)$ and $h_2(x)$. Cuckoo hashing is a form of open addressing algorithm, which is a family of hashing techniques where each cell of a hash table stores a single key-value pair.

The lookup operation verifies if the query item is stored in one of the two buckets. The insertion operation is more complicated. If one of the two buckets is empty, the item is stored there. Otherwise, one of the occupied buckets is selected and the item is kicked out to the other location to free space for the new item. The kicked-out item must be re-inserted possibly kicking out another item there, and so on. The insertion will finish either all items inserted in their buckets, or it can fail forcing the data structure to be rebuilt. The failure occurs when no vacant bucket is found meaning the hash table is full to insert new item.

The Cuckoo hash table operations are exemplified in Figure 6.2. In the insertion

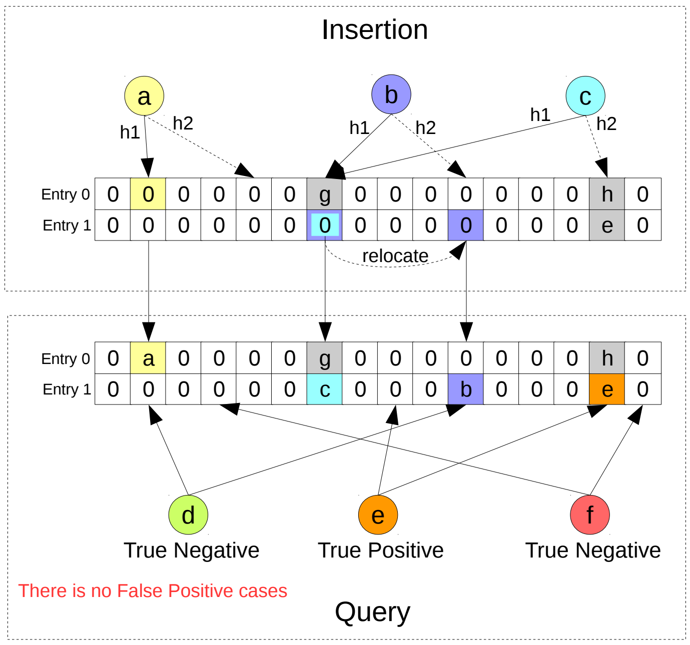


Figure 6.2: Example of Cuckoo Hash Table operations with 16 buckets with 2 entries. Each entry holds an element x which is mapped by $h_1(x)$ and $h_2(x)$ through **cuckoo hashing**. Note that the insertion of the element **c** causes the relocation of the element **b** as all entries of **c** are occupied.

operation, the element b is inserted in the entry 1 of the index $h_1(b)$. As all entries are used when c is inserted, it kicks out the b element relocating it at the entry 1 of the index $h_2(b)$. The query operation is the exact membership query returning no false positive results since it is a type of hash table.

Cuckoo hashing provides high space occupancy by re-organizing the earlier item-placements when new items are inserted. There are extended cuckoo hashing implementations that allow each bucket stores multiple items and use k hash functions to determine k candidate buckets for each item. Configuring the correct parameters of k and bucket size b, the cuckoo hash table space can be 95% filled with high probability [107].

6.1.3 Cuckoo Filter

Cuckoo filter is AMQ data structure based on cuckoo hash tables [107] that stores a small fingerprint for each element belonging to a set, instead of storing a key-value pair. Cuckoo filter consists of an array of m buckets with b entries where each entry stores a f-bit fingerprint. Similar to cuckoo hash table, each item has two candidate buckets specified by hash functions $h_1(x)$ and $h_2(x)$. However, as Cuckoo filter stores the fingerprint of each item from the set, cuckoo hashing cannot be used in Cuckoo filter because it needs to access the original item in the occupied entry to calculate

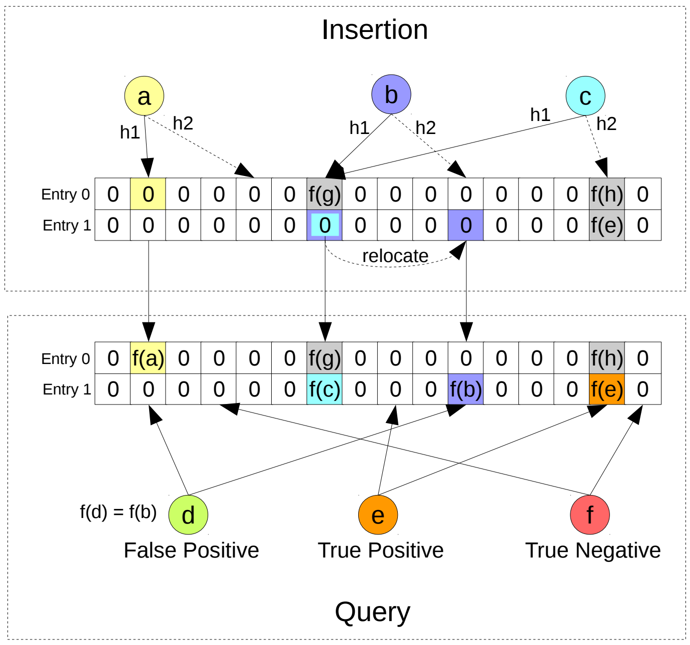


Figure 6.3: Example of Cuckoo filter operations with 16 buckets with 2 entries. Each entry holds the fingerprint $f(x)$ of an element x which is mapped by $h_1(x)$ and $h_2(x)$ through **partial-key cuckoo hashing**. Note that the insertion of the element **c** causes the relocation of the **b**'s fingerprint as all entries of **c** are occupied.

the new entry for relocation during insertion procedure. The cuckoo hashing variant referred to as *partial-key cuckoo hashing* enables Cuckoo filter to dynamically insert new items.

Figure 6.3 shows examples of insertion and query operations performed by Cuckoo filter. In both operations, the partial-key cuckoo hashing is used to calculate the indexes of two candidate buckets from an item x as follows. Considering the fingerprint $f(x)$, the first index will be given by $h_1(x) = hash(x)$ and the second index will be $h_2(x) = h_1(x) \oplus hash(f(x))$. The XOR operation allows to calculate $h_1(x)$ from $h_2(x)$ using the same formula. Thus, when one bucket i is occupied (i might be $h_1(x)$ or $h_2(x)$), the alternate bucket j is directly calculated by $j = i \oplus hash(fingerprint)$ where $fingerprint = bucket[i]$. In the insertion operation, the fingerprint of the input item is inserted into one of the empty entries of the two candidate buckets as occurring with the element a. If no empty entry is found, an occupied entry is selected to store the new fingerprint and the old fingerprint is kicked out to other location and, then, the kicked-out fingerprint is recursively displaced in same manner as in cuckoo hashing method. As shown in the example, primarily the fingerprint of element b is inserted using the index $h_1(b)$. After inserting the element c, the b's fingerprint is relocated at the entry 1 of the index $h_2(b)$. The query operation checks if the fingerprint of the input item matches to any entries of the two candidate buckets. If so, the item is suggested as member of set represented by Cuckoo filter with false positive probability of ϵ. The false positive cases happen when two elements have the same fingerprint and they are mapped to some common bucket as exemplified with the element d.

The false positive probability ϵ can be estimated by approximately $2b/2^f$, where b is the bucket size and f is the number of bits for the fingerprints. This formula can be obtained by observing that, in query operation, the probability of each entry matched against the input fingerprint to be a false positive case is at most $1/2^f$. Therefore, the total of $2b$ entries can be compared from buckets $h_1(x)$ and $h_2(x)$ resulting the total false positive probability hit of $2b/2^f$.

6.1.4 Quotient Filter

Quotient filter is AMQ data structure designed to maintain the locality of data by supporting all functionality of Bloom filter, in addition the merge and delete operations [108]. Quotient filter represents a multi-set S as an array of buckets that store fingerprints corresponding to each element. A hash function $h(x)$ defined as $h : U \to \{0, 1, ..., 2^p - 1\}$ is used to map an element x to a p-bit fingerprint. Considering the false positive probability δ and capacity n (number of elements) in the set, the ideal parameter p can be calculated by $p = \log_2 \frac{n}{\delta}$.

To store the fingerprints, Quotient filter divides $h(x)$ into $h_0(x)$, which is the quotient containing the first q bits (upper bits), and $h_1(x)$, that is the remainder containing the remaining $r = p - q$ bits (lower bits). Quotient filter keeps an array Q of 2^q buckets, each of which can hold one $(r + 3)$-bits: the r-bit remainder plus 3 metadata bits. By inserting an element x, the filter stores the remainder $h_1(x)$ into the home slot $Q[h_0(x)]$. If the slot is already occupied, then a variant of linear probing algorithm is performed using the metadata bits to find a nearby empty slot to store the remainder $h_1(x)$. In collision cases, the remainders are stored contiguously in order and they have the same quotient forming a sequence of occupied slots called *run*. The sequence of consecutive runs is called *cluster*. For creating run and cluster, the linear probing algorithm guarantees three invariants: (1) remainders can only be shifted to right from their home slot, (2) all remainders are stored in order such that if $h(x) < h(y)$, $h_1(x)$ will be stored in a slot before $h_1(y)$ and (3) there are no empty slots between an item and its home slot.

The insertion and lookup operations require a sequential search starting from the home slot to the correct run of the input item as exemplified in Figure 6.4. The run of each slot is determined by analyzing its three metadata bits: *is_occupied*, *is_continuation* and *is_shifted*. The *is_occupied* bit indicates if the slot i is the home slot for any remainder stored in the Q, in other words, if there exists $h(x)$ such that $h_0(x) = i$. The slot marked with *is_continuation* bit stores the remainder which is

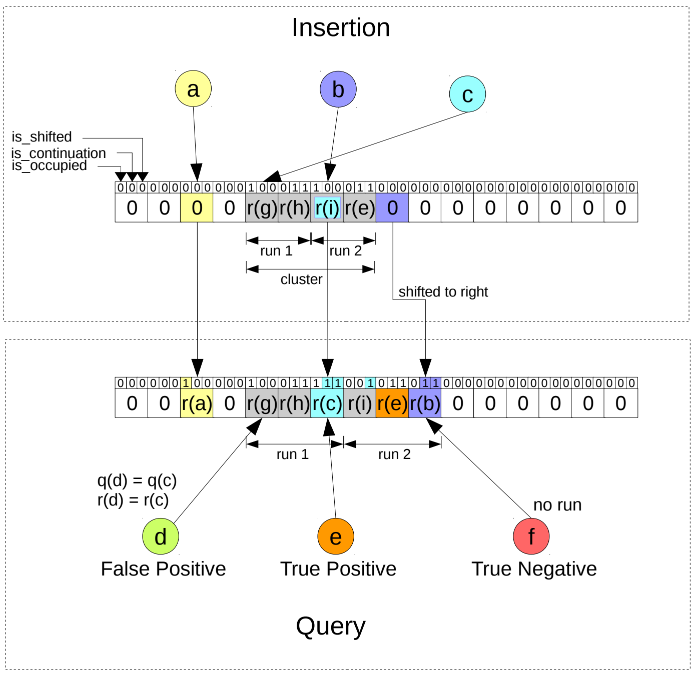


Figure 6.4: Example of Quotient filter operations with 16 buckets. For an element x, the quotient is given by $q(x)$ and the remainder by $r(x)$. The arrows coming from the elements point to their home slot. Note that the insertion of the element **c** shifts to right the elements of run 2, consequently shifting the position of element **b**.

Table 6.1: Meaning of metadata bit combinations from Quotient Filter.

is_occupied	**is_continuation**	**is_shifted**	**Meaning**
0	0	0	Empty slot.
0	0	1	Slot storing the first remainder of a run that has been shifted from its home slot.
0	1	0	Not used.
0	1	1	Slot storing the continuation of a run that has been shifted from its home slot.
1	0	0	Slot storing the first remainder of a run in its home slot.
1	0	1	Slot storing the first remainder of a run that has been shifted from its home slot. There exists a run associated to the home slot which was shifted to right.
1	1	0	Not used.
1	1	1	Slot storing the continuation of a run that has been shifted from its home slot. There exists a run associated to the home slot which was shifted to right.

not the first in its run. Finally, the *is_shifted* bits indicates that the remainder in a slot is not its home slot. The meaning of the possible combinations of these bits is summarized in Table 6.1.

Figure 6.4 exemplifies the Quotient filter operations. In the insertion operation, *is_occupied* bit is always set to 1 in the home slot of the inserted item. Element b's fingerprint is inserted at end of run 2, updating the *is_continuation* and *is_shifted* bits since it is not the first remainder neither it is in its home slot. By inserting element c, all elements of run 2 are shifted to right updating the *is_shifted* bit from the first remainder of run 2 (element i). Also, *is_continuation* and *is_shifted* bits are updated for the c's slot as it is the last remainder from run 1. In the query operation, the false positive cases occur if two elements have the same fingerprint as depicted with element d.

There are efficient implementations of Quotient filter to improve the lookup performance such as Rank-and-Select based Quotient Filter (RSQF) and Counting Quotient Filter (CQF) proposed in [109]. These implementations also support counter operations and provide efficient lookup operations on 64-bit vectors using X86 instruction set. The good data locality of the implementations enables Quotient filter operates efficiently on SSD.

6.2 WiSARD based on AMQ Filters

When adopting a WiSARD architecture, the binary transformation can impact the accuracy and the learning capacity of the model affecting directly its input size, which determines the number of RAMs and the tuple size for each discriminator. Consequently, large RAMs might be required to achieve a good accuracy.

Additionally, it is common to have few memory addresses accessed in comparison with the total positions after training huge RAMs, resulting in several positions with $0's$. To reduce the memory resources by avoiding storage of irrelevant zero positions, WiSARD is extended by replacing RAMs with AMQ filters. The new model is termed AMQ WiSARD which is specialized in one of the AMQ filters discussed in Section 6.1. The key idea is to store a set of tuples mapped to each AMQ filter and test if a given tuple belongs to its corresponding set with certain false positive probability.

6.2.1 Bloom WiSARD - WiSARD based on Bloom Filters

Bloom WiSARD has the same operations as WiSARD. On the training phase, the tuples are inserted into Bloom filters by updating the k bit array positions as illustrated in Figure 6.5. On the classification phase, the tuples are queried into their

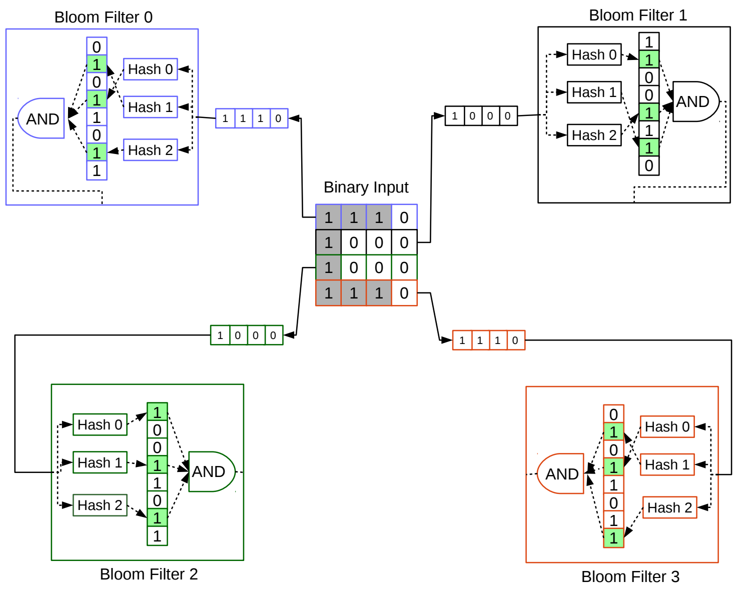


Figure 6.5: Example of training in Bloom WiSARD with 16-bit input, 4-bit tuples and 4 Bloom filters.

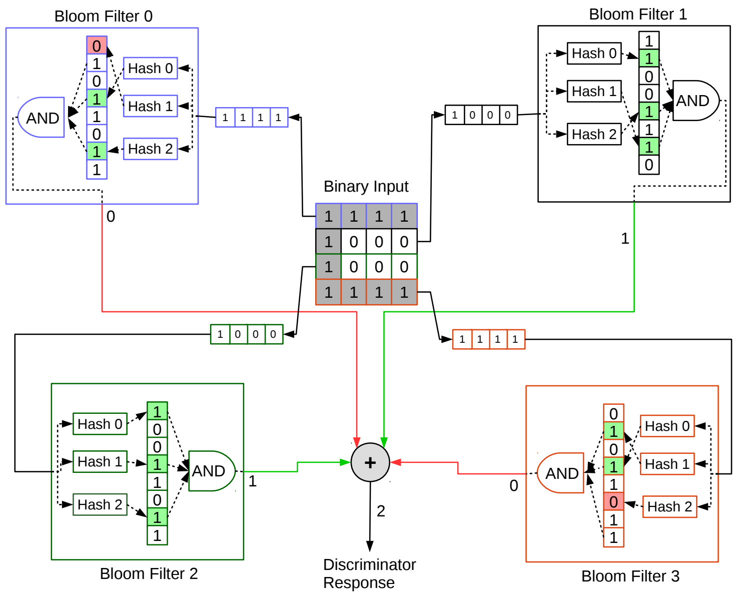


Figure 6.6: Example of classification in Bloom WiSARD with 16-bit input, 4-bit tuples and 4 Bloom filters.

associated Bloom filters returning whether each tuple is a member or not by ANDing all k bit values as presented in Figure 6.6. Similar to WiSARD, the discriminator responses are calculated by summing the N Bloom filter membership results so that the highest response selects the appropriate discriminator to represent the input [110].

In this work, the Bloom WiSARD implementation utilizes a double hashing technique [111] to generate k hash functions in the form: $h(i, k) = (h_1(k) + i \times h_2(k)) \pmod{n}$, where h_1 and h_2 are universal hash functions. MurmurHash function[112] was adopted as seed hashes h_1 and h_2.

6.2.2 Cuckoo WiSARD - WiSARD based on Cuckoo Filters

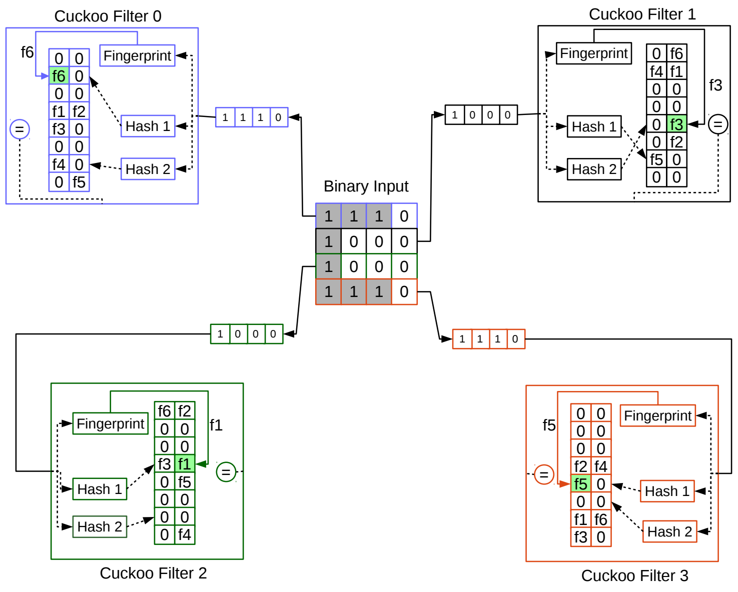


Figure 6.7: Example of training in Cuckoo WiSARD with 16-bit input, 4-bit tuples and 4 Cuckoo filters.

On Cuckoo WiSARD, the tuples are inserted into Cuckoo filters by updating the entries indexed by hash functions h_1 or h_2 as illustrated in Figure 6.7. Cuckoo WiSARD does not support counting operations so that the fingerprints are inserted only once. During classification phase, the tuples are queried into their associated Cuckoo filters returning whether each tuple is a member or not, by comparing the input fingerprint against all fingerprints found at entries associated to the buckets of the query item, as presented in Figure 6.8. The discriminator responses are calculated by summing the N Cuckoo filter membership responses so that the highest response selects the appropriate discriminator to represent the input.

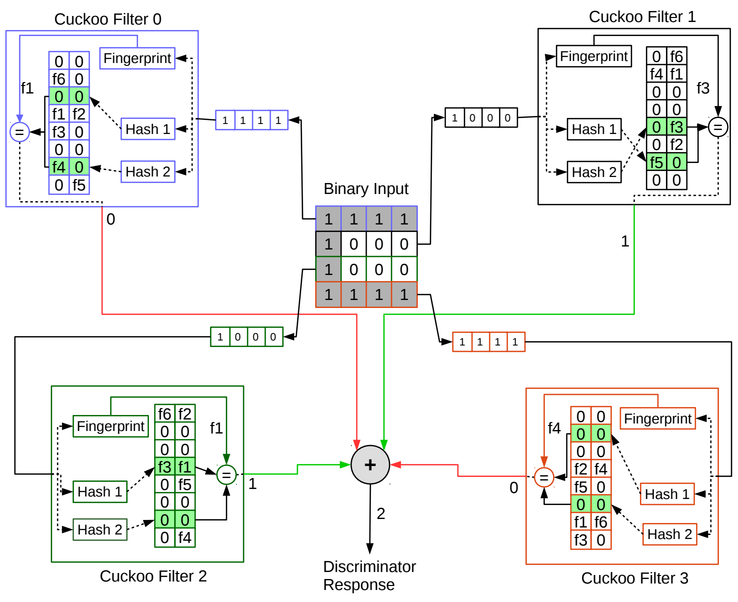


Figure 6.8: Example of classification in Cuckoo WiSARD with 16-bit input, 4-bit tuples and 4 Cuckoo filters.

In this work, the Cuckoo WiSARD implementation utilizes MurmurHash function[112] to generate the fingerprint and the hash functions h_1 and h_2.

6.2.3 Quotient WiSARD - WiSARD based on Quotient Filters

Quotient WiSARD inserts the tuples into Quotient filters by adding the remainder into the correct slot (in order) starting from its home slot as illustrated in Figure 6.9. If the remainder is already included, the insertion operation does not replicate it in other slot. Therefore, it does not support counting operation. On the classification phase, the tuples are queried into their associated Quotient filters returning whether each tuple is a member or not, by comparing the queried remainder against all remainders in the input run as presented in Figure 6.10. By summing the N Quotient filter membership results, it will generate the discriminator response. In the same manner as in WiSARD, the highest discriminator response selects the appropriate discriminator to represent the input.

In this work, the Quotient WiSARD implementation utilizes MurmurHash function[112] to generate the fingerprint which is split into quotient and remainder.

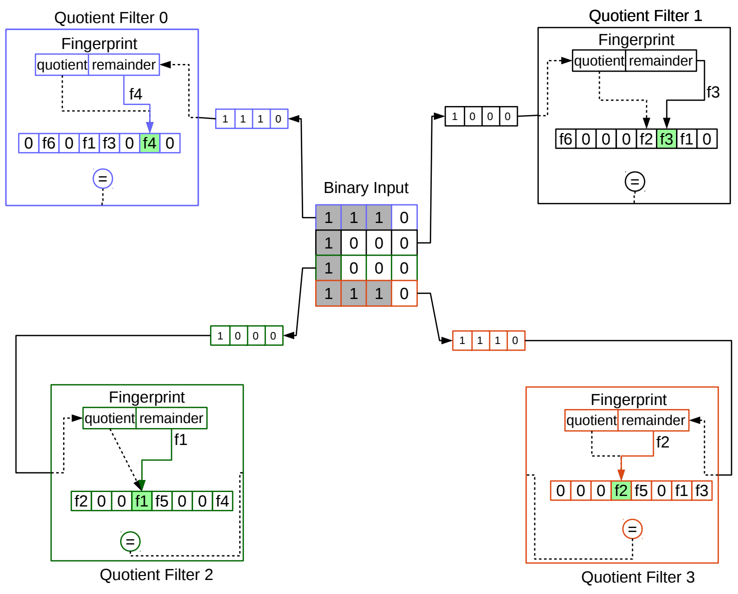


Figure 6.9: Example of training in Quotient WiSARD with 16-bit input, 4-bit tuples and 4 Quotient filters.

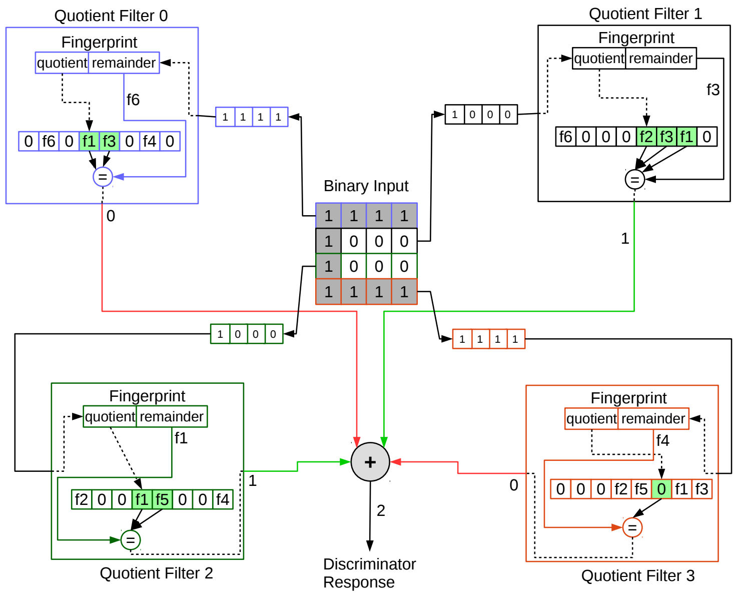


Figure 6.10: Example of classification in Quotient WiSARD with 16-bit input, 4-bit tuples and 4 Quotient filters.

6.3 Experiments and Results

This Section evaluates all proposed models in comparison with two different WiSARD versions: standard WiSARD and dictionary WiSARD, on real-world benchmarks with datasets for binary and multiclass classification. Dictionary WiSARD is implemented with Hash Tables (HT) in place of RAMs to store the tuple values as key-value pair in each HT position with the key representing the memory address [101].

6.3.1 Dataset

Table 6.2: Specification of binary classification datasets.

Dataset	**# Train**	**# Test**	**# Features**
Adult	32,561	16,281	14
Australian	460	230	14
Banana	3,532	1,768	2
Diabetes	512	256	8
Liver	230	115	6
Mushroom	5,416	2,708	22

Table 6.3: Specification of multiclass classification datasets.

Dataset	**# Train**	**# Test**	**# Features**	**# Classes**
Ecoli	224	112	7	8
Glass	142	72	9	7
Iris	100	50	4	3
Letter	13,332	6,668	16	26
MNIST	60,000	10,000	784	10
Satimage	4,435	2,000	36	6
Segment	1,540	770	19	7
Shuttle	43,500	14,500	9	7
Vehicle	564	282	18	4
Vowel	660	330	10	11
Wine	118	60	13	3

A subset of binary classification and multiclass classification datasets used in [113] and MNIST database [114] were selected to extensively evaluate the new WiSARD models. Most of the problems were taken from UCI public repository [115] and they have different characteristics in terms of number of samples, number of classes and number of features. Table 6.2 and Table 6.3 tabulate the parameters of the binary and multiclass classification datasets, respectively.

For datasets that do not provide the training set and testing set in separated files, the same methodology applied in [113] was adopted: the single data is randomly shuffled and partitioned in 3 parts, such that 2/3 are used as training set and 1/3 composes the testing set.

6.3.2 Experimental Setup

The experiments were performed on CPU machine with an Intel Core i7-8700(3.20GHz) processor and 64GB of RAM running Ubuntu Linux 16.04 operating system. All WiSARD versions were implemented using C++11 language with single thread providing a library to be utilized in Python, while the experiments were implemented in Python. To convert the input attributes to binary format, all binary attributes are concatenated using thermometer to transform the continuous attributes and one hot encoding to transform categorical attributes. The input size, number of RAMs and tuple size varying according to dataset. These parameters are kept to all WiSARD versions. In particular for Bloom WiSARD, the capacity is empirically selected from each dataset and m and k are calculated through the formulas presented in Section 6.1.1.

6.3.3 Accuracy, Performance and Memory Consumption Results

For all datasets, each model was run 20 times and the mean of accuracy, training time and testing time were obtained with negligible standard deviation. Bloom filters are setup with 10% of false positive probability in Bloom WiSARD model. The results for Cuckoo WiSARD and Quotient WiSARD are obtained with the best configuration in number of buckets and fingerprint size which achieves competitive accuracy in comparison with the other models. Tables 6.4 and 6.5 show the results for binary classification and multiclass classification datasets, respectively.

Overall, Bloom WiSARD, Cuckoo WiSARD and Quotient WiSARD achieved comparable accuracy, training time and testing time results while consuming less memory than standard and dictionary versions. In some datasets, the training time of standard WiSARD was slower as it has the larger RAMs and was not able to take full advantage of cache locality. Dictionary WiSARD and the standard WiSARD did not obtain the same accuracy as they have different pseudo-random mappings for each run. AMQ WiSARD models' memory consumption are reduced up to 6 orders of magnitude (Adult and Letter) compared to standard WiSARD and approximately 7.7 times (Banana) when compared with dictionary WiSARD. The three models of AMQ WiSARD have competitive memory consumption where each one obtains smaller memory than the others depending on the dataset. Since Bloom WiSARD achieves interesting results, its memory resources can be further reduced by increasing the false positive rate.

Table 6.4: Accuracy, training time, testing time and memory results for Binary Classification problems.

Dataset	**WNN**	**Accuracy (%)**	**Training (s)**	**Testing (s)**	**Memory (KB)**
Adult	WiSARD	72.1	3.1284	0.7509	8978432
	Dict WiSARD	72.14	1.4093	0.8741	392.08
	Bloom WiSARD	70.69	1.3859	0.8585	48.164
	Cuckoo WiSARD	74.77	1.2634	0.8208	17.125
	Quotient WiSARD	75.56	1.805	1.1999	17.125
Australian	WiSARD	83.83	0.0016	0.0008	4096
	Dict WiSARD	83.54	0.0017	0.0010	11.777
	Bloom WiSARD	83.17	0.0017	0.0011	1.875
	Cuckoo WiSARD	83.19	0.0025	0.0013	2.187
	Quotient WiSARD	83.3	0.0019	0.0012	4.0
Banana	WiSARD	87.07	0.0365	0.0207	13312
	Dict WiSARD	87.03	0.0397	0.0243	23.730
	Bloom WiSARD	86.59	0.0418	0.0266	3.047
	Cuckoo WiSARD	87.45	0.0398	0.0249	3.25
	Quotient WiSARD	87.14	0.0422	0.0271	6.5
Diabetes	WiSARD	68.83	0.0009	0.0005	2048
	Dict WiSARD	69.41	0.001	0.0006	6.992
	Bloom WiSARD	67.69	0.001	0.0006	0.469
	Cuckoo WiSARD	67.42	0.001	0.0006	0.375
	Quotient WiSARD	67.15	0.0011	0.0007	1.0
Liver	WiSARD	59.0	0.001	0.0005	5120
	Dict WiSARD	58.48	0.0011	0.0006	6.113
	Bloom WiSARD	58.48	0.001	0.0007	2.344
	Cuckoo WiSARD	58.83	0.0013	0.0006	1.875
	Quotient WiSARD	58.69	0.001	0.0006	2.5
Mushroom	WiSARD	1.0	0.0355	0.02	8192
	Dict WiSARD	1.0	0.0394	0.0244	20.850
	Bloom WiSARD	1.0	0.0406	0.0257	3.75
	Cuckoo WiSARD	99.51	0.0384	0.0238	2.0
	Quotient WiSARD	99.99	0.041	0.0257	4.0

Table 6.5: Accuracy, training time, testing time and memory results for Multiclass Classification problems.

Dataset	**WNN**	**Accuracy (%)**	**Training (s)**	**Testing (s)**	**Memory (KB)**
Ecoli	WiSARD	79.2	0.00039	0.00037	7168
	Dict WiSARD	79.55	0.0004	0.00041	5.312
	Bloom WiSARD	79.46	0.00039	0.00053	3.281
	Cuckoo WiSARD	79.02	0.00039	0.00044	2.625
	Quotient WiSARD	79.42	0.00039	0.00044	3.5
Glass	WiSARD	72.99	0.0021	0.002	51968
	Dict WiSARD	71.53	0.002	0.002	20.61
	Bloom WiSARD	72.64	0.0019	0.0025	23.789
	Cuckoo WiSARD	73.19	0.0018	0.0022	19.031
	Quotient WiSARD	72.22	0.0019	0.0024	25.375
Iris	WiSARD	97.6	0.00011	0.00007	1536
	Dict WiSARD	98.0	0.00012	0.00007	0.732
	Bloom WiSARD	97.7	0.00011	0.00008	0.703
	Cuckoo WiSARD	98.3	0.0001	0.00007	0.562
	Quotient WiSARD	97.9	0.0001	0.00007	0.75
Letter	WiSARD	84.77	0.17791	0.11618	10223616
	Dict WiSARD	84.71	0.0533	0.17826	122.002
	Bloom WiSARD	84.55	0.0517	0.17122	54.844
	Cuckoo WiSARD	84.02	0.05116	0.1639	117.0
	Quotient WiSARD	83.76	0.05254	0.17998	78.0
MNIST	WiSARD	91.65	3.5748	0.2478	9175040
	Dict WiSARD	91.55	0.59067	0.3376	1447.803
	Bloom WiSARD	91.68	0.56471	0.29368	819.219
	Cuckoo WiSARD	91.2	0.53919	0.28843	280.0
	Quotient WiSARD	91.5	1.13211	1.35538	280.0
Satimage	WiSARD	85.7	0.03479	0.02605	27648
	Dict WiSARD	85.5	0.037198	0.03771	65.273
	Bloom WiSARD	84.93	0.03893	0.03947	12.656
	Cuckoo WiSARD	83.7	0.03812	0.03618	27.0
	Quotient WiSARD	84.75	0.03797	0.0412	27.0
Segment	WiSARD	93.88	0.00642	0.00569	17024
	Dict WiSARD	93.27	0.00676	0.0076	7.554
	Bloom WiSARD	93.85	0.007132	0.00893	7.793
	Cuckoo WiSARD	93.06	0.007519	0.00899	7.793
	Quotient WiSARD	93.36	0.00686	0.00819	8.312
Shuttle	WiSARD	87.04	0.08499	0.04932	8064
	Dict WiSARD	86.82	0.08813	0.05845	4.819
	Bloom WiSARD	86.86	0.09726	0.08036	3.691
	Cuckoo WiSARD	86.01	0.08884	0.06437	1.969
	Quotient WiSARD	86.8	0.08972	.06535	3.937

Table 6.6: Accuracy, training time, testing time and memory results for Multiclass Classification problems (continuation).

Dataset	WNN	Accuracy (%)	Training (s)	Testing (s)	Memory (KB)
Vehicle	WiSARD	67.76	0.00235	0.00156	9216
	Dict WiSARD	66.42	0.002561	0.00195	17.734
	Bloom WiSARD	66.52	0.002499	0.00218	4.219
	Cuckoo WiSARD	66.98	0.00445	0.00279	5.625
	Quotient WiSARD	66.79	0.00256	0.00225	9.0
Vowel	WiSARD	88.20	0.00146	0.00171	1144
	Dict WiSARD	88.11	0.00179	0.00273	16.221
	Bloom WiSARD	86.41	0.00171	0.0032	8.379
	Cuckoo WiSARD	88.45	0.0031	0.00408	7.519
	Quotient WiSARD	88.29	0.00164	0.00264	6.875
Wine	WiSARD	93.33	0.000417	0.000239	4992
	Dict WiSARD	93.08	0.000423	0.000258	4.526
	Bloom WiSARD	92.75	0.000382	0.000293	2.285
	Cuckoo WiSARD	93.83	0.000376	0.000257	1.219
	Quotient WiSARD	94.0	0.000382	0.000268	2.437

6.3.4 Bloom WiSARD: False Positive Rate vs. Accuracy vs. Memory Analysis

In Section 6.3.3, the false positive rate of Bloom filters were fixed to 10% using the formulas presented in Section 6.1.1. In contrast to traditional use of Bloom filter, where it needs to ensure the correct query responses with high probability, Bloom WiSARD does not require low false positive rate because even if a tuple is erroneously returned as member of a Bloom filter, the model is not compromised and can still improve the generalization capability of the system. In order to evaluate the potential of Bloom WiSARD, the accuracy and memory consumption are evaluated for different configurations of the false positive rate. For all datasets, the rate is varied from 10% to 90%.

The results are presented in Figure 6.11. The memory consumption and accuracy decrease according to the increase of the false positive probability. Overall, the accuracy is kept acceptable until the 50% false positive rate that is decreased in average 1.77% with worst case about 6.1% (Australian). Also, the memory is reduced about 3.16 times in comparison against 10% of false positive rate. In addition, the number of hash functions for each Bloom filter is reduced from 4 (10%) to 2 (50%) resulting in slight increase of speed up on training and classification phase.

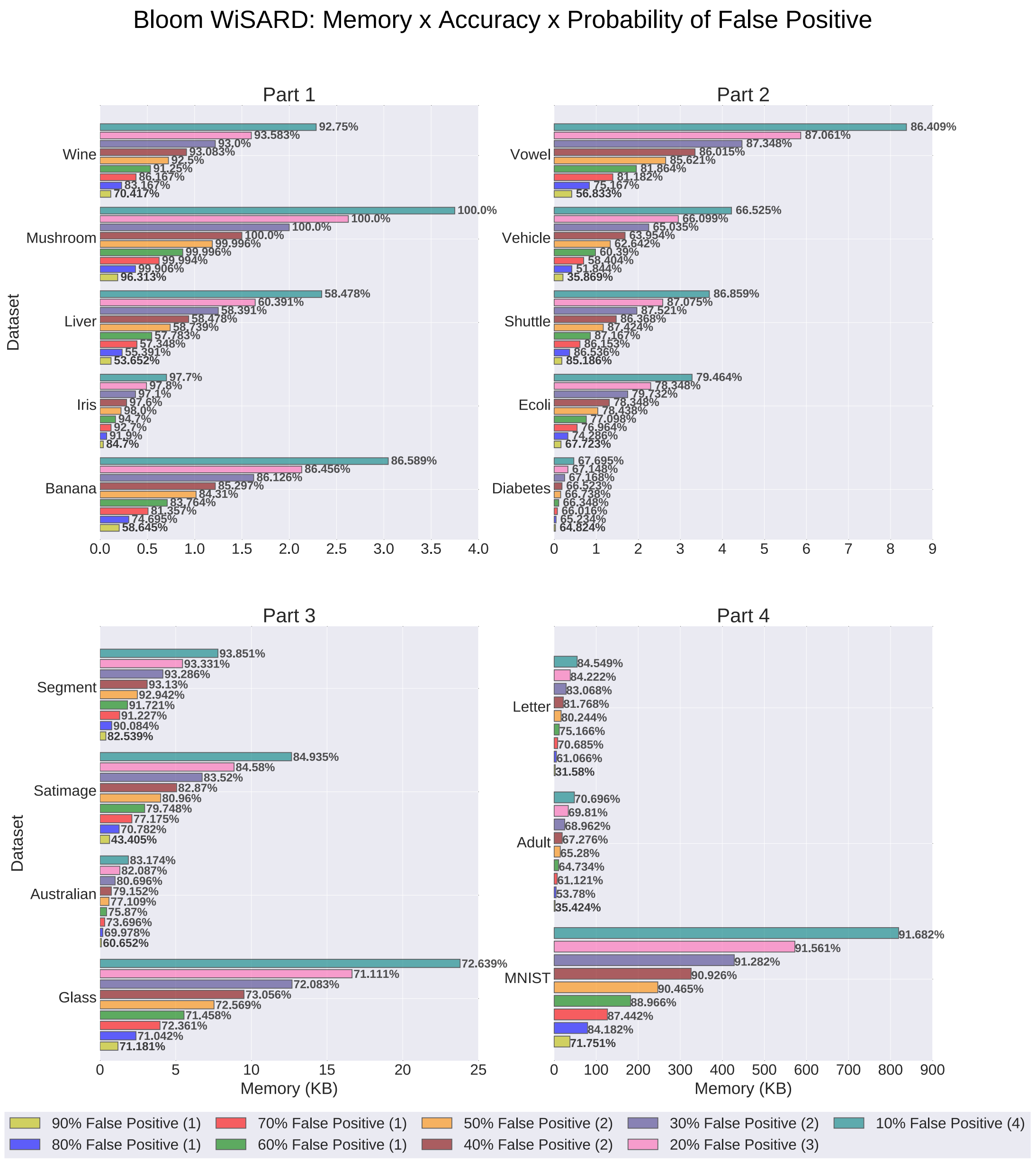


Figure 6.11: Accuracy and memory consumption results when varying the false positive rate of Bloom WiSARD. In the legend, the number of hash functions is shown in parentheses at end of each false positive rate. The accuracy is shown at right side of each bar.

Table 6.7: Accuracy and memory results for AMQ WiSARD compared to 50% false positive probability. The memory and accuracy results from Bloom WiSARD are obtained with 50% false positive probability. The remaining results are selected from configurations that approximate the Bloom WiSARD accuracy.

Dataset	**Bloom WiSARD**		**Cuckoo WiSARD**		**Quotient WiSARD**	
	Acc%	***M(KB)***	***Acc%***	***M(KB)***	***Acc%***	***M(KB)***
Adult	65.28	14.717	73.34	8.562	68.58	2.141
Australian	77.11	0.594	80.59	0.937	79.28	0.5
Banana	84.31	1.016	87.45	3.25	87.64	1.625
Diabetes	66.74	0.156	66.54	0.25	66.11	0.5
Ecoli	78.44	1.039	79.02	2.625	78.84	0.875
Glass	72.57	7.533	71.60	12.687	71.18	6.344
Iris	98.00	0.223	98.30	0.562	97.90	0.375
Letter	80.24	16.758	79.99	58.5	82.18	39
Liver	58.74	0.742	58.83	1.875	58.13	1.25
MNIST	90.46	246.641	90.15	210	90.93	140
Mushroom	99.99	1.187	99.51	1	99.84	2
Satimage	80.96	4.008	80.26	10.125	80.48	6.75
Segment	92.94	2.468	92.61	6.234	93.02	4.156
Shuttle	87.42	1.169	87.11	3.076	87.15	5.906
Vehicle	62.64	1.336	63.05	2.25	64.75	2.25
Vowel	85.62	2.653	86.64	6.875	85.41	3.437
Wine	92.5	0.724	93.25	0.762	92.25	2.437

6.3.5 AMQ WiSARD: Accuracy vs. Memory Analysis

As analyzed in Section 6.3.4, Bloom WiSARD could obtain acceptable accuracy and reduced memory when the false positive rate is configured to 50%. Unlike Bloom WiSARD, the other AMQ filters were not generated using false positive probability as parameter. In order to compare the AMQ WiSARDs, various results were obtained from Cuckoo and Quotient WiSARD by varying the tags per bucket, tag bits (fingerprint and remainder) and number of buckets parameters. The complete results are displayed in Appendix B.

To evaluate AMQ WiSARDs with Bloom WiSARD with 50% false positive rate, the results with competitive accuracy and low memory consumption are selected and tabulated in Table 6.7. Bloom WiSARD achieves the best memory consumption in 11 datasets, while Cuckoo WiSARD achieves in 1 and Quotient WiSARD in 5 datasets. In comparison against Bloom WiSARD 10% of false positive rate, the memory is reduced in average 2.03 times with the best case about 5.62 (Adult) for Cuckoo WiSARD and in average 3.47 times with the best case about 22.5 (Adult) for Quotient WiSARD.

It indicates that all AMQ models have competitive potential to reduce the memory maintaining good accuracy. Bloom WiSARD can be easily configured by using the formulas presented in Section 6.1.1 providing good learning capability. The other models are required to be empirically analyzed with different configurations in order to find reduced memory requirements. Even Bloom WiSARD is practical to use, the other models are interesting approaches to be considered as they rely on few universal hash functions that could be critical in hardware implementation. Both Cuckoo and Quotient filter implementations are based on their standard model. There are optimized versions elaborated to reduce the memory requirements such as the semi-sorting buckets in Cuckoo filter which saves 1 bit per fingerprint [107] and Rank-and-Select based Quotient Filter (RSQF) which uses 2.125 metadata bits per slot (standard model uses 3 metadata bits per slot) [109].

6.4 Concluding Remarks

WiSARD is a powerful WNN model based on RAM memory that can be easily implemented in hardware and real-time systems. Nevertheless, certain applications require a considerable amount of memory to achieve good learning capabilities becoming impracticable to implement it in current technology. Alternative structures like dictionary are required to implement the RAM nodes and enable the use of the model. In this work, AMQ WiSARD model is proposed to address the memory consumption of WiSARD by implementing RAM nodes as AMQ filters. By using these filters, the memory resources are widely reduced by allowing occurrences of false positives. The main AMQ filters: Bloom filter, Cuckoo filter and Quotient filter, are used to construct the novel models. Although false positives are very detrimental in certain applications, for pattern recognition purposes it was experimentally found that they can build robustness into the system (as dropout does to deep neural networks). The results show that the models provide good accuracy, training and testing time. When Bloom WiSARD is setup to 10% false positive rate, it consumes up to 6 orders of magnitude less resources than standard WiSARD and about 7.7 times less resources than WiSARD implemented with dictionaries. In addition, increasing the false positive rate to 50% results in 3.16 times less memory and average of 1.77% decreased accuracy compared to 10% configuration. Similar results are achieved by Cuckoo WiSARD and Quotient WiSARD over standard WiSARD and dictionary WiSARD. In comparison with approximate accuracy of Bloom WiSARD with 10% false positive rate, the memory is reduced about 2.03 and 3.47 times, respectively. Future work will focus on extending the AMQ filter operations such as frequency counts of elements stored, in order to enable AMQ WiSARD to use improved techniques such as DRASiW. Hardware implementation for AMQ WiSARD models will also be evaluated.

Chapter 7

Conclusion

HW-assisted security solutions have been deployed to support the security of data which traffics over higher layers of system architecture and is out of user control. Particularly, this thesis explores optimizations regard *Physically Unclonable Functions* (PUFs) and *Dynamic Information Flow Tracking* (DIFT) fields which are adequate for implementing *HW-assisted Random Generation*, *HW-assisted Malware Detection* and *HW-assisted Pointers Violation Prevention* technologies.

PUFs are promising hardware security primitives to develop authentication and key generation systems while they offer highly resistance against invasive attacks. However, Strong PUF implementations are still inadequate as model building attacks based on machine learning algorithms can clone them with high accuracy. This thesis proposed various novel Strong PUF designs based on WiSARD, a simple Weightless Neural Network (WNN) model, which increase the resistance against machine learning attacks. Furthermore, an entropy source is combined to WNN PUF architectures to create reliable Strong PUF keeping the high machine learning attack resistance. As result, minimum reliable Strong PUFs with $< 65\%$ machine learning accuracy can be constructed by using only 32 bits of reliable Weak PUF.

High volume manufacturing (HVM) of PUFs requires post-manufacturing real-time analysis to test if the adequate uniqueness property is achieved among the produced chips. As most of solutions proposes offline analysis, this thesis elaborated strategies to optimize an online design-for-test methodology based on Multi-Index Hashing (MIH) structure. MIH is a fast search technique for binary codes in sub-linear time in Hamming space. Nonetheless, it requires the data be copied into multiple hash tables. By investigating the field of k-nearest neighbor (kNN) search for Hamming space, the methods of distance free computation search and hamming weight theorem are adapted in order to reduce the memory space without impact the performance.

Recently, binary compression techniques have been employed to compact the high-dimensional data into binary codes to make feasible kNN search applications.

Since MIH can be used as solution for kNN search using binary codes, this thesis also formulated a co-processor design to accelerate data-intensive MIH operations. It is composed of specialized Hamming distance calculation and Odd-even Merge Sort Network circuits. The proposed hardware was implemented on low-cost FPGA running at 100 MHz and evaluated for kNN search problem obtaining interesting speedup in comparison with embedded ARM processor when dataset is large. The co-processor can be applied properly in the online PUF testing with MIH and the other online kNN search solutions based on Hamming distance calculation such as Hamming Weight Tree.

DIFT is an appealing technique to track sensitive information propagation and impede its leakage. Nevertheless, an adversary can explore implicit flows propagation to evade explicit-propagation based tracking. This thesis developed a lightweight Nested Implicit Flow Tracking (NIFT) technique able to track implicit flow originated from control dependencies. In contrast to other implicit DIFT techniques, it is effective to hold implicit flow along multiple nested branches. A formal verification is provided to validate the correctness of NIFT design. In addition, a new taint propagation rule was defined to tag data under control dependencies when it come from immediate assignment instructions. The experiments showed that the new taint propagation mitigates the over-tainting problem and NIFT was precise in catching implicit information flow with low overhead in terms of performance and memory consumption.

As explored for Strong PUF designs, WiSARD is a simple WNN model that can be efficiently implemented in hardware. However, certain pattern recognition applications require an impractical large amount of memory to achieve acceptable learning results. Finally, this thesis proposed new WiSARD models based on Approximate Membership Query (AMQ) data structure. By using AMQ structures, memory resources are significantly reduced at the expense of allowing false positives when verifying if a given data was already recorded in the network. The experiments showed that the proposed models achieve competitive accuracy, training time and testing time consuming up to 6 orders of magnitude less resources than standard WiSARD and about 7.7 times less resources than dictionary-based WiSARD when they are setup to 10% false positive rate. Increasing the Bloom WiSARD's false positive rate to 50%, the memory resources is reduced about 3.16 times and the accuracy is decreased approximately 1.77% in comparison with 10% false positive configuration. Cuckoo WiSARD and Quotient WiSARD reduced the memory about 2.03 and 3.47 times, respectively, in comparison against approximate accuracy of Bloom WiSARD with 10% false positive rate. Additionally, the experimental results indicate that AMQ structures can build robustness into the system for pattern recognition purposes.

Since pattern recognition applications are increasingly used on IoT environments, the WiSARD model is an appealing solution to run on these systems. In the future, AMQ WiSARD models can be implemented in hardware in order to evaluate the required resources for each design. As WiSARD was also applied for security purposes to create new Strong PUF design, one work could analyze the resilience of proposed models by including interference of noise on training or testing set. In the context of hardware security, new mechanisms might be proposed to create more resilient WNN model enabling it as secure solution in critical pattern recognition applications such as pedestrian detection in autonomous car. Overall, the knowledge of robust neural circuit is interesting making it possible to be used as hardware component to support HW-assisted security technologies for any kind of applications running on ubiquitous computing.

Appendix A

Hardware Similarity Search with Multi-Index Hashing

In current decade, artificial intelligence and machine learning applications have received widespread attention so that big companies such as Google [116], Microsoft [117] and ARM [118] have invested on novel hardware architectures to accelerate such domain specific problems. Furthermore, data-intensive applications have led to architectural trends as Von-Neumann CPUs are not adequate to offer higher throughput and lower latency required for such applications.

Similarity search is a class of data-intensive machine learning problems that is important for several applications like computer vision [119], image retrieval [120] and natural language processing [121]. Formally, the problem consists of searching the most similar data from a database given a query as input. A well-know variation is the *k-nearest neighbors* (kNN) search problem which selects only until k of the most identical contents.

High dimensional real-valued feature vectors produced by modern applications bring a challenge to appropriately manage them in practical time. To overcome this barrier, binary code generation techniques have been broadly explored to improve the efficiency in similarity search domain by indexing high-dimensional vectors into the Hamming space preserving similarity in the original space [122–127]. Since the points are drastically reduced into binary code space, choosing the appropriate indexing structure is critical as it impacts accuracy and performance of kNN search. Recent work has shown that hash based structures achieve significant speed-up opportunities especially when distance search is done in Hamming Space [128].

Multi-index hashing (MIH) is a fast search technique to retrieve nearest neighbors in Hamming space [76]. To speed up the search, it relies on multiple hash tables to store indexes mapped by disjoint sub-strings partitioned from each binary code. Thereby, parallel near neighbor searches on these sub-strings are able to find out all neighbors for the complete binary code. This Chapter proposes an efficient hardware co-processor design to accelerate similarity search on MIH structure. The Hamming distance calculation and partially sort of the candidate neighbors, that are executed in each hash table, were identified as data-intensive and practical operations to be accelerated in the hardware. The Register Transfer-Level (RTL) architecture is specified in Verilog hardware description language and implemented in the programmable logic of a low-cost Xilinx FPGA (XC7Z020). Performance, circuit-area and power requirement of the proposed accelerator are analyzed and compared to embedded ARM processor in order to validate it for low-cost environment such as IoT applications.

A.1 K-Nearest Neighbors Search Algorithm

Nearest neighbor search problem can be defined as retrieving the closest values of a query q in a set P containing n objects represented as points in a space. When P has high-dimensional objects, the search is challenging by requiring efficient storage/indexing strategies and distance calculation. A variation of the problem is the k-nearest neighbors search where the number of found closest points is limited by k.

Recent works have proposed different types of indexing strategies on high-dimensional data such as: hierarchical k-means, kd-trees, multi-probe locality sensitive hashing (MPLSH) and multi-index hashing (MIH). The idea of these techniques is to create a data structure by compacting the search space and, consequently, drastically reducing the query time. Hierarchical k-means generates clusters of a database organized in a tree structure, where the similar entries are holding in the leaves that are accessed by a query with closest contents [129]. Identically, kd-trees constructs a tree structure to index the clusters of a database which are formed by a random slice of the database [130]. Multi-probe locality sensitive hashing (MPLSH) is based on multiple hash tables, where each hash location stores similar entries [131]. Unlike the conventional hash functions which aim to avoid collisions, MPLSH holds hash functions to maximize collisions so that queries are mapped to hash table locations with identical content. Multi-index hashing (MIH) stores the binary codes into multiple hash tables indexed by disjoint parts of data as detailed in Section 2.5 [76, 132]. Even binary codes sacrifice accuracy during the quantization process, it is cheap and straightforward to implement in hardware.

These techniques are based on some distance calculation between the stored data and query to determine the level of similarity. There are several metrics for distance like Euclidean distance, Hamming distance [133], cosine similarity, learned distance metrics [134], Manhattan distance and Jaccard similarity. Although Euclidean distance is the most common, each metric is better for specific index strategy. Hamming distance is prominent to measure distance in binary codes by taking advantage from the MIH structure [76].

A.2 The Accelerator Architecture

Considering the MIH structure presented in Section 2.5, MIH hardware co-processor is proposed to accelerate hamming distance calculation and partial sorting operations as soon as the sub-strings are looked-up from each hash tables. In other words, the co-processor receives an input stream of binary codes and produces an ordered list of candidates for each hash table.

A.2.1 Top View

The kNN search using MIH requires the delineation of a Hamming search radius r in order to retrieve the k nearest neighbors from each hash table. Since different search queries with radius r may produce more candidates than k, the radius parameter must not be fixed. The search process starts from radius parameter $r = 0$ iterating it until the top-k similar neighbors are found. In each radius iteration i, the i-neighbors are validated as true neighbors and sent to an odd-even merge sort pipeline to partially sort the already found exact candidates from each hash table. At the end, the top k nearest candidates are selected from the final ordered list generated by merging all ordered list corresponding to each hash table.

Figure A.1 illustrates how the MIH co-processor can be used to accelerate the kNN search for a 16-bit binary query partitioned in 4 sub-strings and a Hamming search radius $r = 0$ (one sub-string represents itself a neighbor). Each neighbor of sub-string is a key to access a MIH hash table entry containing a list of indexes. These indexes point out to the 16-bit binary code stored in the database. First the complete neighbor data is recovered from the database through the indexes stored in each entry of the hash. Then, the accelerator calculates the Hamming distances between the query and the data streamed from the database. Independently, the distances are ordered for each hash entry and are ranked among the hash tables until to obtain the top-k candidates.

At certain conditions, kNN search in MIH may require multiple search radius for all queries to find the most k similar neighbors. Moreover, huge databases are likely to map several data onto the same hash table entry. Based on those facts, MIH data-intensive computation comprises the hamming distance calculation and the partial sorting of candidates performed in each hash table. As data-intensive processing applications can benefit from higher memory bandwidth, the proposed MIH accelerator implements specialized circuits for hamming distance calculation and partial sorting. Several instances of the co-processor can be connected to each

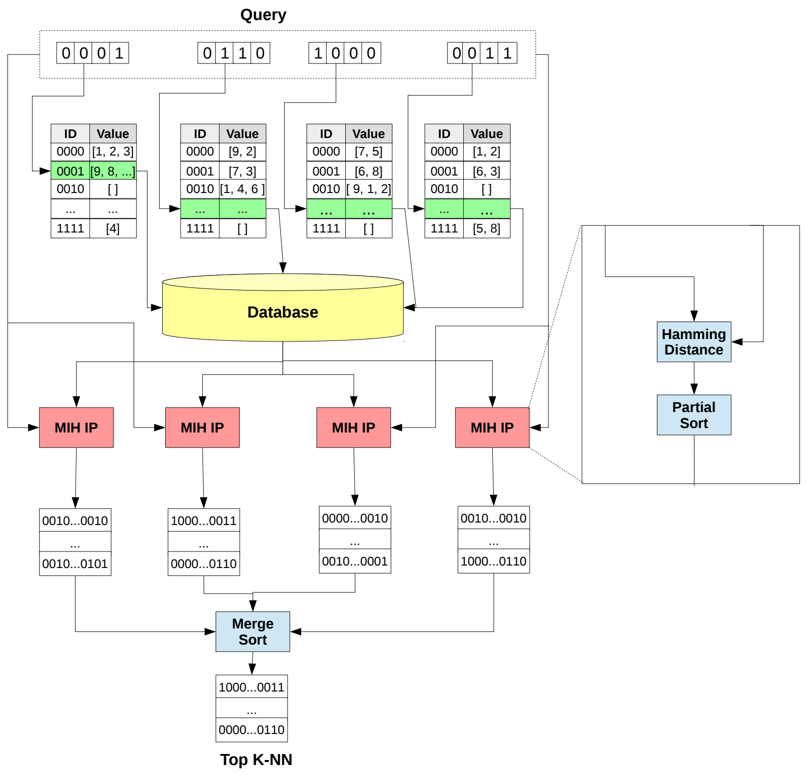


Figure A.1: K-nearest neighbor search with Multi-Index Hashing (MIH) hardware accelerator (MIH Intellectual Property - IP). It presents a search for a 16-bit binary query partitioned in 4 sub-strings and a Hamming search radius $r = 0$.

hash table in order to quickly produce the sorted small lists with the top nearest neighbors. Due to this, the merge sort process can be optimized considering that the candidates from all hash tables are ordered by the query similarity.

A.2.2 Accelerator Design

To support the data-intensive execution phases, the MIH accelerator was designed to communicate with external systems via Advanced eXtensible Interface (AXI) Stream protocol, which is part of the Advanced Microcontroller Bus Architecture (AMBA4). The accelerator has two main cores: the AXI Stream Slave, which receives the query and the stream of neighbor data, and AXI Stream Master, which sends back the sorted distance list with the corresponding neighbor indexes. For purposes of simplicity, the architecture is described to support 64-bit binary string hamming distance calculation and 64 sorted neighbor distances and indexes. Different configurations require the extension of the architecture components in order to fulfill the specification. Figure A.2 depicts the MIH accelerator architecture.

The AXI Stream slave core holds a double buffer to store the sorted distances and indexes. Initially, the first 64-bits are stored in the *query* register while the subsequent data is stored in the *rx_data* register. As soon as data is received, the hamming distance is calculated and the data index (receiving order) is sent to the sorter component. Both Hamming distance and sorter components are detailed in Section A.2.3. When the first buffer is full (64 distances), the core waits until the buffer ordering is completed to start the insertion in the second buffer. To speed up the candidate selection, the distances are only kept in the second buffer if it is less than the last distance stored in the first buffer. That rule avoids to sort far neighbors by early eliminating the false nearest neighbors. When the second buffer is full, the core starts the merge process between both buffers to organize the smallest distances into the first buffer and reinitialize the second buffer. The merge process is detailed in Section A.2.4. After merging, the logic comes back to insert small distances on the second buffer by comparing to first buffer. Those processes continue until the last neighbor data is received.

AXI Stream master core starts after slave core has finished. The sorted distances are sent to the host after all sorted indexes have been submitted. Since a double buffer is used to store the indexes and distances, master core handles two pointers to choose the smaller distances between the buffers until it sends the k ($k = 64$) candidates.

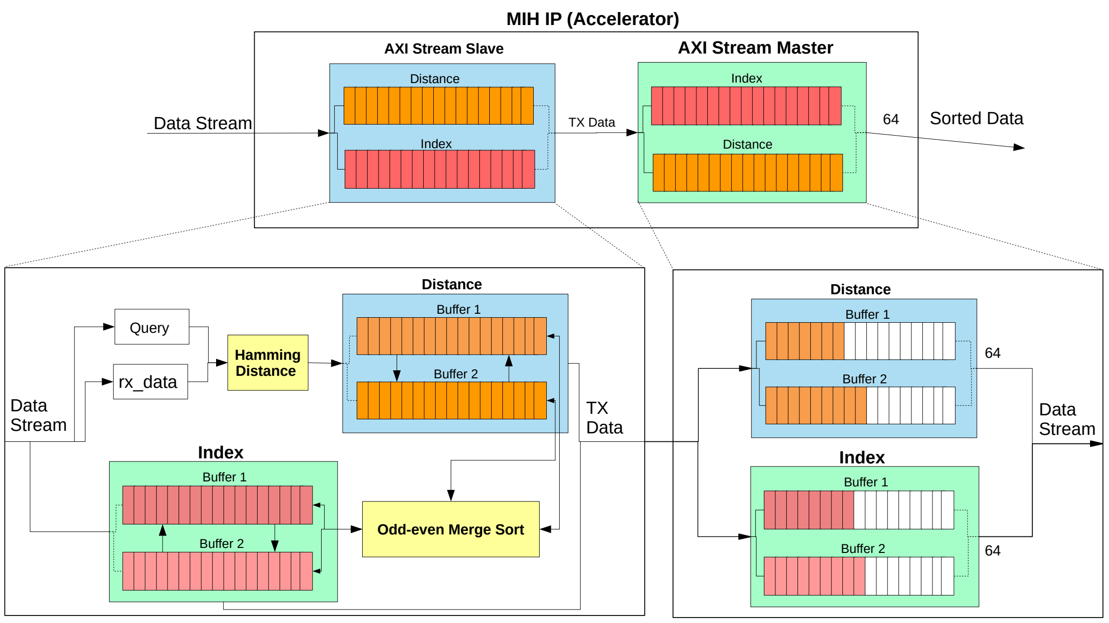


Figure A.2: Overview of the MIH Accelerator. The AXI Stream Slave interface receives the query and the stream of neighbor data (64-bit each), while the AXI Stream Master sends back the top-64 sorted distances and the corresponding neighbor index.

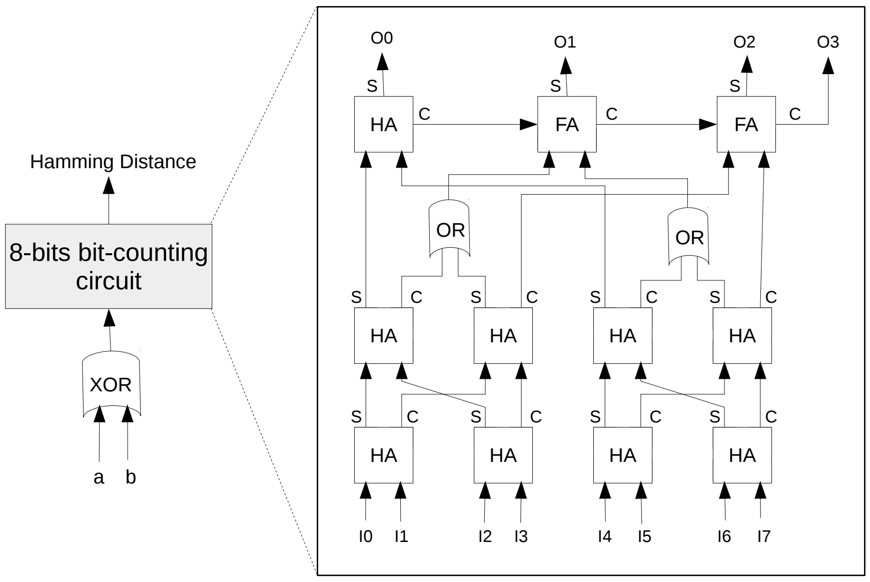


Figure A.3: 8-bits Hamming Distance counting component, as proposed in [4].

A.2.3 Specialized Components

The efficiency of the co-processor comes from the fast specialized circuit to calculate Hamming distance and the sorting network to sort the indexes and distances received by slave core. Hamming distance between binary codes a and b can be efficiently obtained by counting number of $1's$ in $a \oplus b$. There are several techniques to perform bit-counting operation, which is known as *popcount* (population counts) operation. For this work, the hamming distance component is implemented to take only one cycle with the embedded bit-counting circuit proposed by Dalalah *et al.* [4] as their circuit requires fewer resources and achieves low delay than other hardware solutions, as shown in Figure A.3.

Following the method proposed by Dalalah *et al.* [4], the bit-counting circuit can be recursively generated by extending the base 8-bits counting circuit. The base circuit has 3 layers connecting *Half Adder* (HA) and *Full Adder* (FA) components with *OR* logic gate. Table A.1 summarizes the number of gates required to build such bit-counting circuit for different input sizes.

Table A.1: Bit-counting consumption. Each Full Adder consumes 5 gates and Half Adder 2 gates.

Bit length	**# FAs**	**# HAs**	**# OR gates**	**Total Gates**
8	2	9	2	30
16	7	19	4	77
32	18	39	8	176
64	41	79	16	379

The partial sorting component is implemented through sorting network algorithms that are very efficient to implement and run on hardware. The circuit is formed of horizontal wires and vertical comparators with the unsorted elements come from left side. A fixed number of comparisons and swap operations are executed along the wires to sort the elements two-by-two in each stage, keeping smaller elements on the upper output wire while the larger elements are placed on lower output wire. Several sorting network implementations have been evaluated on FPGA by Mueller *et al.* who conclude that Batcher's odd-even merge sort network is the best cost benefit in terms of hardware resource and throughput [135]. Batcher's algorithm describes a recursive non-adaptive procedure to generate the sorting network with a number of items with size of power of 2, taking $O(\log(n)^2)$ depth to sort n elements utilizing $O(n \log(n)^2)$ resources. As that algorithm is adequate to sort a short fixed sequence of items, the odd-even merge sort network is implemented as the partial sorting component of the proposed MIH accelerator.

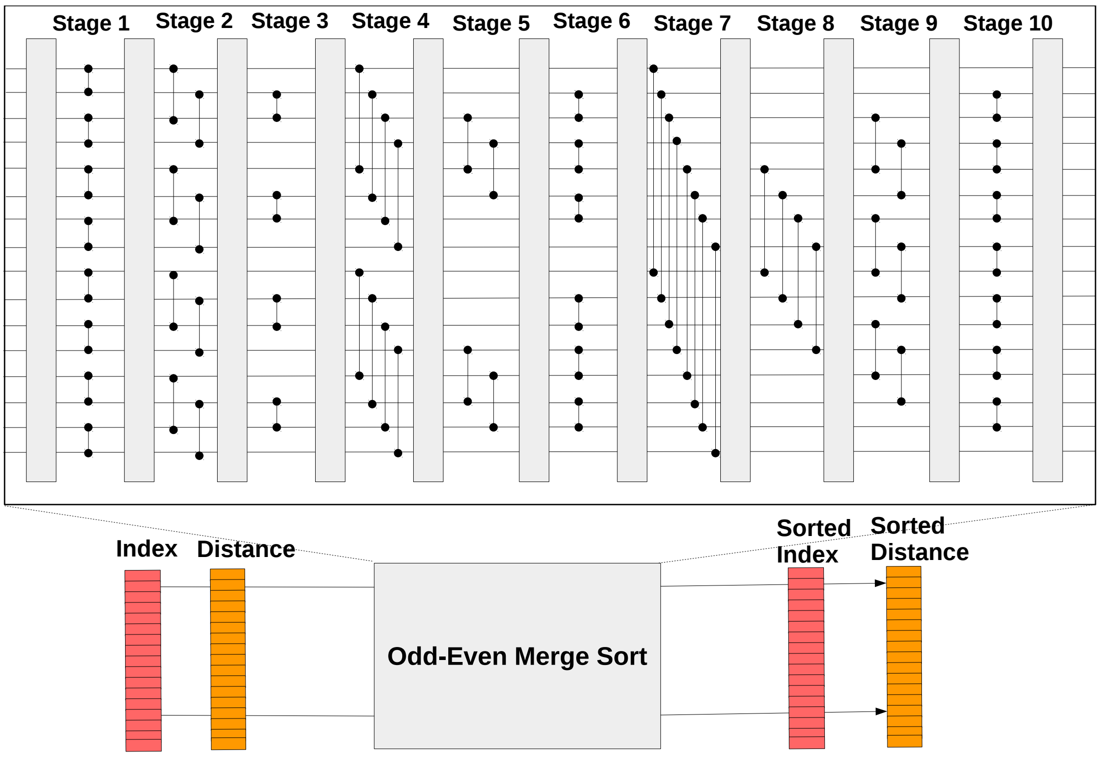


Figure A.4: Odd-even merge sort component, exemplifying the sort network with 16-bit inputs, 16-bit outputs and 10 pipeline stages.

Figure A.4 exemplifies the odd-even merge sort network with 16-bit inputs and 16-bit outputs where the sorting is finished at last stage (stage 10). It is adapted to perform index ordering synchronized with distance order checking and implemented as a pipeline circuit which receives the distance and index buffers together at first stage. By using Batcher's algorithm to sort 64 elements, the pipeline consists of 21 stages with the total of 543 comparators as tabulated in Table A.2.

Table A.2: Batcher's odd-even merge sort pipeline consumption.

Bit length	**# Pipeline Stages**	**# Comparator-and-Swap**
4	3	5
8	6	19
16	10	63
32	15	191
64	21	543

A.2.4 Buffer Merge Strategy

When buffer 1 and 2 are full, the merge between both buffers is executed to store the first sorted elements between them into buffer 1, while buffer 2 is re-initialized with maximum input values to receive the next distances from the input stream.

Buffer merge process is executed in three steps as demonstrated in Figure A.5. Both buffers have 64 items totaling 128. When merge is finished, buffer 1 will hold the smaller 64 distances while the remaining (stored in buffer 2) will be discarded. In step 1, the first 32 elements from buffer 1 and 2 are concatenated and sent to the odd-even merge sorting network, which takes 21 cycles to complete. Similarly, the last 32 elements from both buffers are concatenated and sent to the sorting network component in the step 2. As it also takes 21 cycles to finish, the sorted results from step 2 are obtained one cycle after step 1 has finished due to the pipeline circuit implementation. To summarize, steps 1 and 2 are completed in 22 cycles. Since buffer 1 and 2 are initially sorted, the first 32 items merged by the step 1 represent the true order, that is, they are the 32 smallest elements between buffer 1 and 2. Finally, the last 32 elements resulted from step 1 are concatenated with the first 32 distances sorted from step 2 and, afterwards, they are submitted taking more 21 cycles. At the end, the first 32 items are updated as last 32 elements into buffer 1 which contains the 64 smallest distances between initial buffer 1 and 2. Thus, the buffer merge process always takes 43 cycles.

A.3 Experiments

All experiments and results presented in this Section were executed and implemented on ZYNQ XC7Z020-1CLG400C Xilinx FPGA attached on the Pynq-Z1 board, which contains 512 MB DDR RAM and several I/O interfaces. The Zynq FPGA architecture embeds a Cortex-A9 multi-core processor around its reconfigurable logic. The proposed MIH accelerator specified in Verilog is synthesized and implemented through Xilinx Vivado tools, version 2018.1. In the Pynq boards, Linux Pynq Image v2.3 was installed to execute Python 3.0 and implement the kNN search application based on MIH to access the hardware accelerator. As the boards are not attached

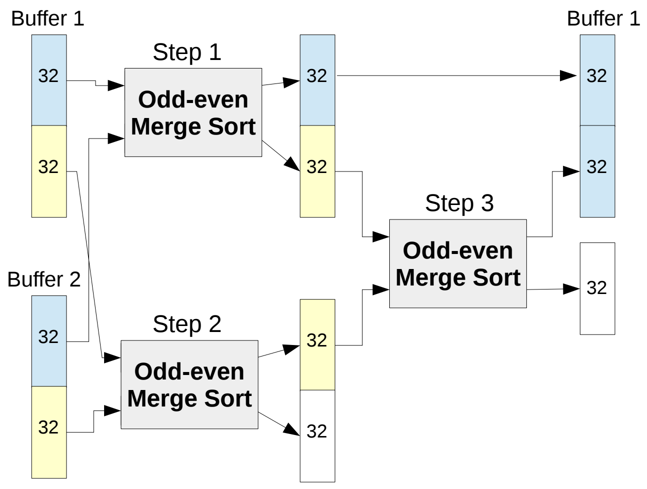


Figure A.5: Buffering merge process of buffer 1 and 2. The process is concluded in 43 cycles - 22 cycles corresponding to steps 1 and 2 plus 21 cycles comes from step 3.

to any machine, they work as a standalone system emulating embedded/edge environment by addressing a low power system with limited computational resources.

A.3.1 Data sets

To evaluate the proposed MIH co-processor implementation, the experiments were conducted on two datasets: 80M 384D GIST descriptors from 80 million tiny images [119] and 1B 128D SIFT descriptors from BIGANN dataset[136]. Since these datasets are not binary, the hyperplane Locality Sensitive Hashing (LSH) [137] was used to generate 64-bit binary codes. Each experiment requires three set of data: a training set to adjust the LSH parameters, a base set to populate MIH structure and the query set that contains the query data. For GIST, the whole data is randomly split into $300K$ items for the training set, 1000 items for query set and the remaining for the base set in similar way as applied in [76]. SIFT descriptors are already divided into training, base and query set.

To binarize the datasets, the mean of training set is subtracted with the input data to normalize the input. Next, the dot product is calculated between a set of coefficients from normal distribution and the normalized input, finalizing with the quantization. These generated binary data were saved into a 32GB microSD card which is plugged into the Pynq board and later read by the application.

A.3.2 Performance

Three versions of the MIH kNN search are implemented to analyze the potential performance of the MIH accelerator: C (entirely in software), Python (entirely in software) and Python with MIH IP. All versions utilize only one core of the dualcore ARM Cortex-A9 host processor. Since the dataset is binarized to 64-bit binary codes, MIH is configured to hold 4 and 8 hash tables. Due to the limited resources provided by the given Zynq FPGA chip (xc7z020), parts of the datasets are used: SIFT10K, SIFT1M, GIST10K, GIST100K and GIST1M. SIFT10K and SIFT1M are already available on the SIFT dataset. For the GIST dataset, 10K, 100K and 1M points are randomly picked from the binary base set corresponding to GIST10K, GIST100K and GIST1M, respectively. Table A.3 summarizes the dataset configurations used to measure the performance.

Table A.3: Dataset configurations.

Dataset	**# training**	**# base**	**# query**
SIFT10K	$25,000$	$10,000$	100
SIFT1M	$100,000$	$1,000,000$	$10,000$
GIST10K	$300,000$	$10,000$	$1,000$
GIST100K	$300,000$	$100,000$	$1,000$
GIST1M	$300,000$	$1,000,000$	$1,000$

For each configuration, the mean of 10 runs is obtained to calculate the speedups. Figure A.6 presents the performance of the MIH accelerator compared to Python code running on the ARM host processor. For all datasets, the MIH accelerator achieves better results than the corresponding Python version. The number of hash tables in MIH influences the distribution of indexes that are sent to the accelerator and the number of radius searches to find the k nearest neighbors. In smaller datasets, the average number of indexes submitted to MIH is less than 64. Since the co-processor uses a sorting network, as presented in Section A.2.3, even receiving less than 64 data, it takes 21 cycles to complete the ordering. The MIH configuration with 8 hash tables obtained best results with the best case accelerating up to 33 times.

Figure A.7 shows the performance of the MIH accelerator compared to C code running on the ARM host processor. Smaller datasets result in slowdown, because the co-processor always takes 21 cycles to sort small list of indexes. Furthermore, the data distribution in 8 hash tables configuration creates many small lists of indexes for each hash table location, so that the C version also takes advantage of the cache hit. The best results were obtained with 4 hash tables, with the best case presenting approximately 19 times speedup.

The performance analysis shows that the MIH accelerator potentially improves the performance of kNN search applications, especially considering that the Zynq ARM processor operates at 667 MHz, while the accelerator operates at 100 MHz. The bottleneck of MIH accelerator is strictly related to the number of indexes stored in each hash table location which impacts the partial sorting operation. A large amount indexes result in great speedup. As the experiments were performed using a

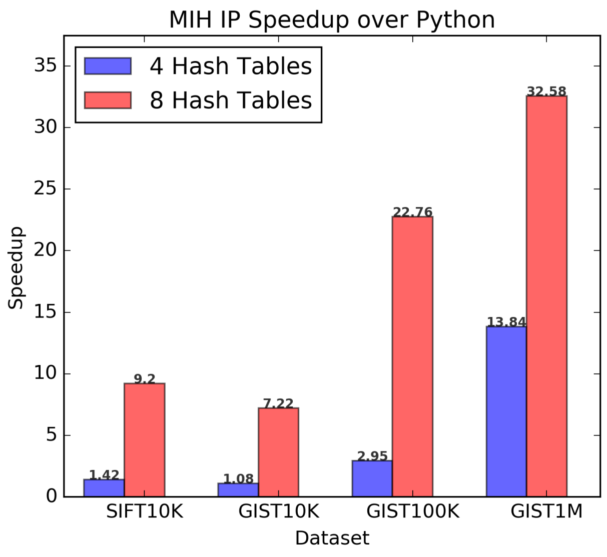


Figure A.6: Speedup of MIH IP compared to Python version.

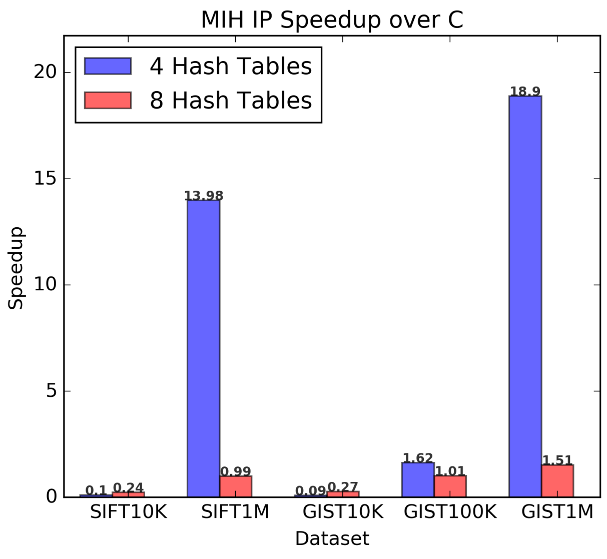


Figure A.7: Speedup of MIH IP compared to C version.

single accelerator for all hash tables due to the limited number of resources available on the Zynq FPGA, the results could be improved if one dedicated MIH accelerator could be instantiated for each hash table.

A.3.3 Utilization Cost

This Section presents the results regarding resource consumption on the implementation of the Hamming distance and sorting algorithm on the Zynq FPGA. The values of the Direct Memory Access (DMA) are also taken into account, since it is needed due to Axi-Stream communication protocol.

Table A.4: Application utilization on the Pynq board.

Resource	**Utilization**	**Available**	**Utilization %**
LUT	47228	53200	88.77
LUTRAM	1934	17400	11.11
FF	61564	106400	57.86
BRAM	18	140	12.86
BUFG	1	32	3.13

Table A.4 presents the resource consumption divided into Look-Up Tables(LUTs), Look-Up Tables RAMs Flip-Flops(FF), Block RAMs (BRAM) and BUFG which is responsible for generating clock. FF is the most used resources considering the raw value and not the percentage. As multiple pipelines are used in the co-processor, data are forced to be stored in small register-like variables which might be mapped onto Flip-Flops. The BRAM resources amounted to 12.86% utilization due to store multiple 64-bit values for the partial sorting component. LUTs are the most used resources ratio-wise achieving 88.77% as there are many logical and arithmetic operations occurring in parallel as seen in Figures A.2 and A.3, which means the hardware needs to be replicated instead of reused. Also, the LUT resource is the main building block of several architecture components.

A.3.4 Power

This Section presents power requirement estimations generated by Vivado power analysis report. The analysis was executed on *vectorless* mode with a default toggle rate set to 12.5% and static probability set to 0.5. Although the *vectorless* power analysis is not accurate, it gives a reasonable estimation about the overall circuit power consumption.

Figure A.8 displays the average power consumption in Watts, separated by FPGA components. Total consumption is equivalent to 1.63 Watts per cycle, which can be considered small when compared to common general purpose ASICs. PS7,

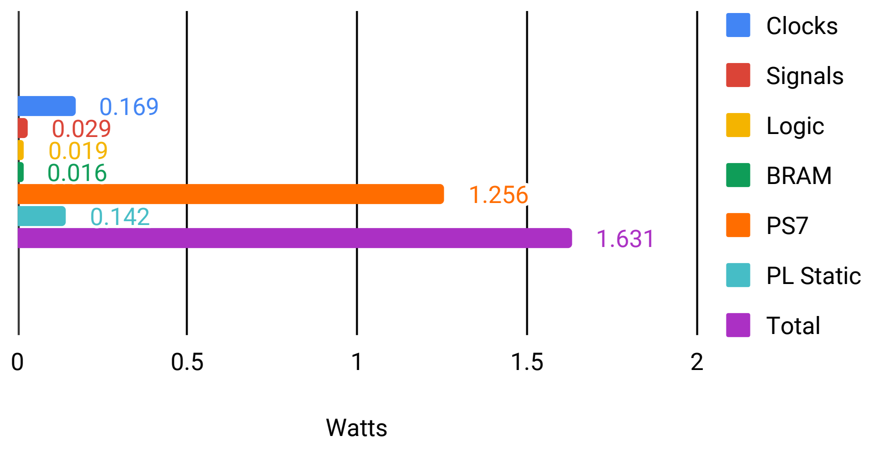


Figure A.8: Vivado 2018.1 Power consumption report based on *vectorless* mode, with a default toggle rate set to 12.5% and static probability set to 0.5.

which is the processing system (ARM), has the most consumption overall, while other components have a very small impact.

A.4 Concluding Remarks

Discussion

As the main goal of this work is to provide a design of hardware accelerator to improve kNN performance using MIH structure, the accelerator can be easily modified to support the variations of this structure such that Distance-Computation-Free search proposed by Song *et al.* [90]. MIH is one type of Inverted Multi-Index (IMI) structure which has been widely researched to provide efficiency in similarity search applications [90, 138, 139], where the key idea is to replace the standard quantization techniques by inverted indices with product quantization [138]. Eghbali *et al.* have elaborated Hamming Weight Tree (HWT) structure [92] which builds a tree structure to map the query with bucket represented as a leaf, containing similar neighbors. Even as in MIH, HWT might be used for kNN search and consists of calculating HD for each candidate into the leaf and sort them to find the top-k most similar neighbors among the leaves. The MIH co-processor can be straightforwardly applied to work with HWT structure.

Moreover, MIH accelerator supports fixed configurations in term of number of candidates to return and to sort by using the sorting network component. The setup of this work uses 64 items to be able to run the empirical experiments discussed in Section A.3.2. Typically, the kNN search uses 100-NN until 1000-NN neighbors parameters. Adjusting these parameters embedded in the sorting network is impracticable due to resource constraints. Kobayashi *et al.* have proposed a sorting accelerator that combines Sorting Networking and Merge Sorter Tree algorithms with a data compression mechanism [140]. Upgrading the sorting network component with their sorting accelerator, it would be possible to increase the number of parameters enabling 1000-NN search in the hardware design.

Overall the results show that the MIH co-processor has the potential to accelerate kNN search consuming around $0,375$ Watts (excluding the ARM processor) and running at 100 MHz. In future works, novel sorting network circuits applicable into kNN search domain will be investigated. Additionally, the proposed design will be replicated on FPGA devices containing more resources to test multi-parallel execution with hamming distance calculations to complement the experiments. Finally, more experiments will be performed taking into consideration Intel and AMD processors.

Conclusion

In order to keep acceptable accuracy when performing kNN search applications, binary compression techniques have been proposed to compact the high-dimensional data into binary codes. Multi-index hashing provides fast indexing solution by mapping database indexes m times into m hash tables, which enables the search for binary codes in sub-linear run-time in Hamming space. Its data-intensive tasks are focused in Hamming distance calculation and the ordering over neighbor candidates from each hash table.

In this work, a MIH hardware co-processor relying on specialized Hamming Distance calculation and Odd-even Merge Sort Network circuits is proposed. Details of FPGA implementation using AMBA4 AXI Stream protocol are discussed and the synthesis results are presented. The proposed hardware running at 100 MHz frequency could achieve interesting speedup in comparison with embedded ARM processor when the dataset is large. The complete co-processor occupies around 88% of the total Look-Up Tables and has low power consumption (around 0.375 Watts, excluding the ARM processor) in a small-sized, low-cost FPGA. The power consumption estimation is based on Vivado's *vectorless* mode, with a default toggle rate set to 12.5% and static probability set to 0.5. In the future, architecture's power consumption will be meticulously measured in order to get more precise power consumption results.

Finally, it is important to emphasize that although the proposed MIH accelerator was implemented and evaluated using a FPGA chip, it could also be implemented as an Application-Specific Integrated Circuit, especially because its RTL architecture is already specified in Verilog hardware description language.

Appendix B

Complete Accuracy Results of Cuckoo and Quotient WiSARD

This Appendix presents the complementary results regarding the Cuckoo WiSARD and Quotient WiSARD presented in Chapter 6. Since both models have more parameters than Bloom WiSARD, the accuracy results with various settings are organized by dataset in the following Sections. The intention is that the reader can fully verify the potential of AMQ WiSARD models. In Section 6.3.5, Bloom WiSARD with 50% false positive rate is compared to Cuckoo and Quotient WiSARD with approximate accuracy results selected from the experiments presented in this Appendix.

B.1 Adult

The results of adult dataset are presented in Figure B.1 and Figure B.2 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

B.2 Australian

The results of australian dataset are presented in Figure B.3 and Figure B.4 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

B.3 Banana

The results of banana dataset are presented in Figure B.5 and Figure B.6 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

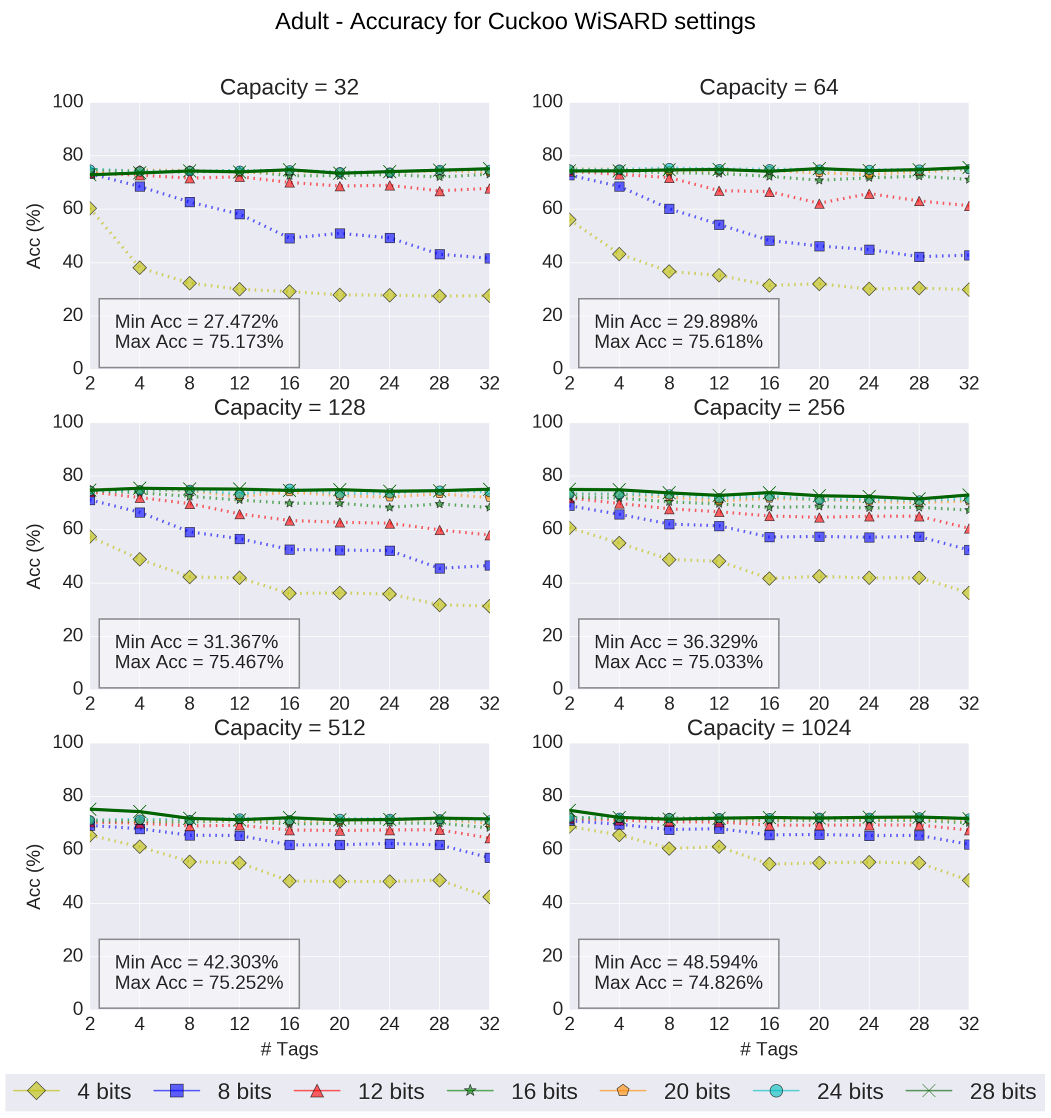


Figure B.1: Accuracy results of Adult dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

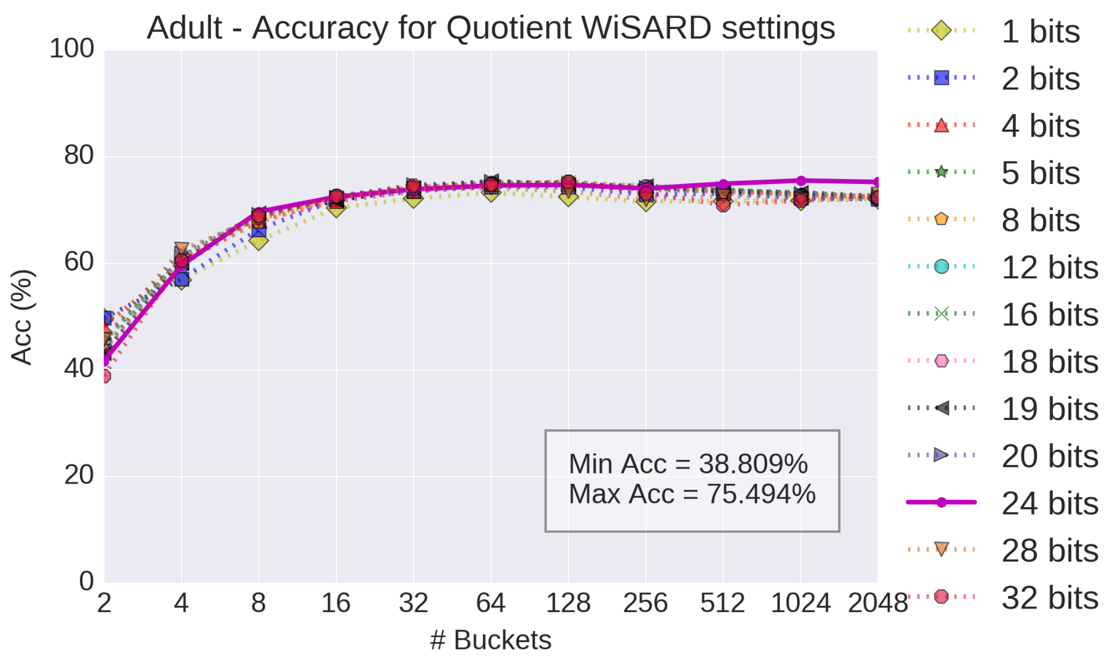


Figure B.2: Accuracy results of Adult dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

B.4 Diabetes

The results of diabetes dataset are presented in Figure B.7 and Figure B.8 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

B.5 Ecoli

The results of ecoli dataset are presented in Figure B.9 and Figure B.10 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

B.6 Glass

The results of glass dataset are presented in Figure B.11 and Figure B.12 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

B.7 Iris

The results of iris dataset are presented in Figure B.13 and Figure B.14 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

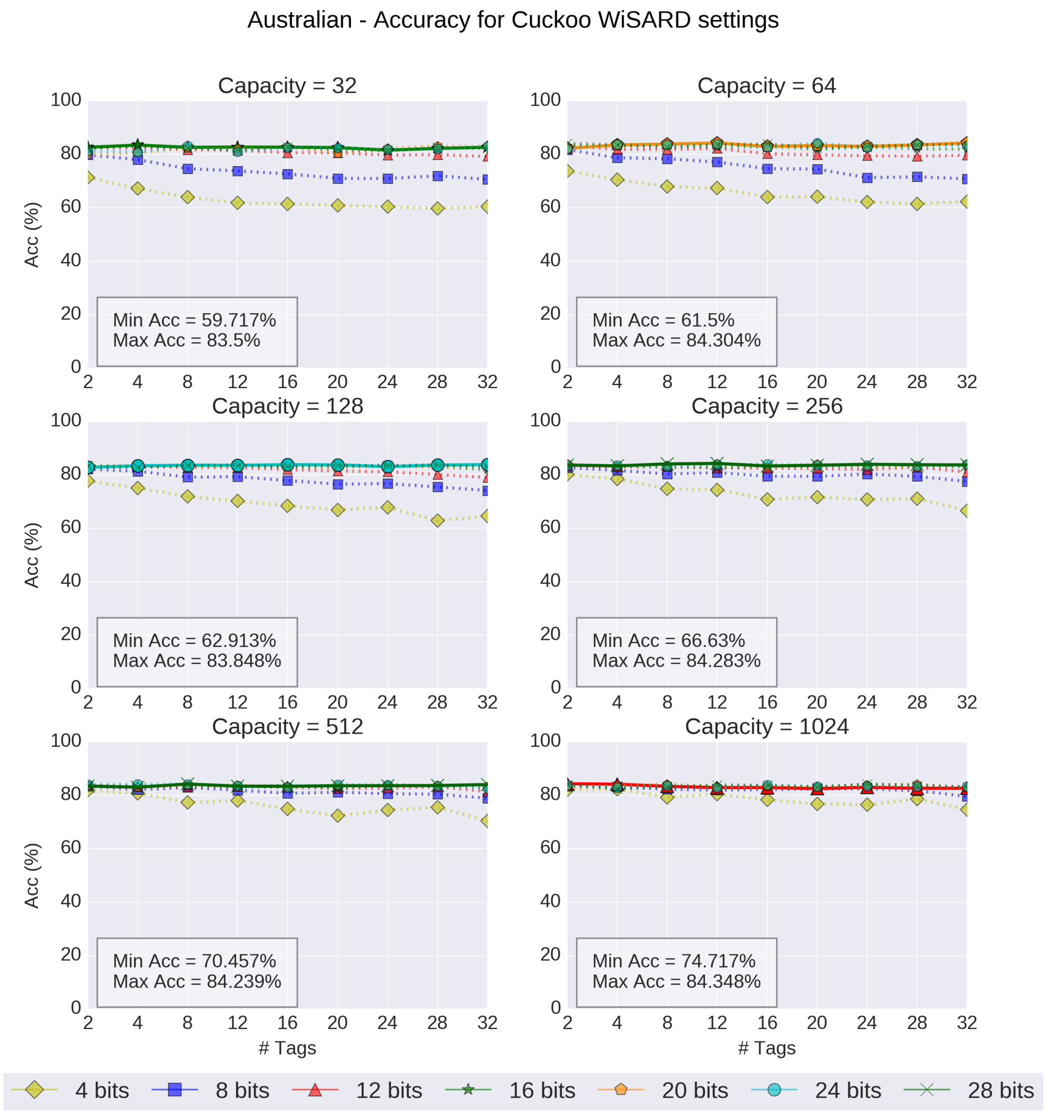


Figure B.3: Accuracy results of Australian dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

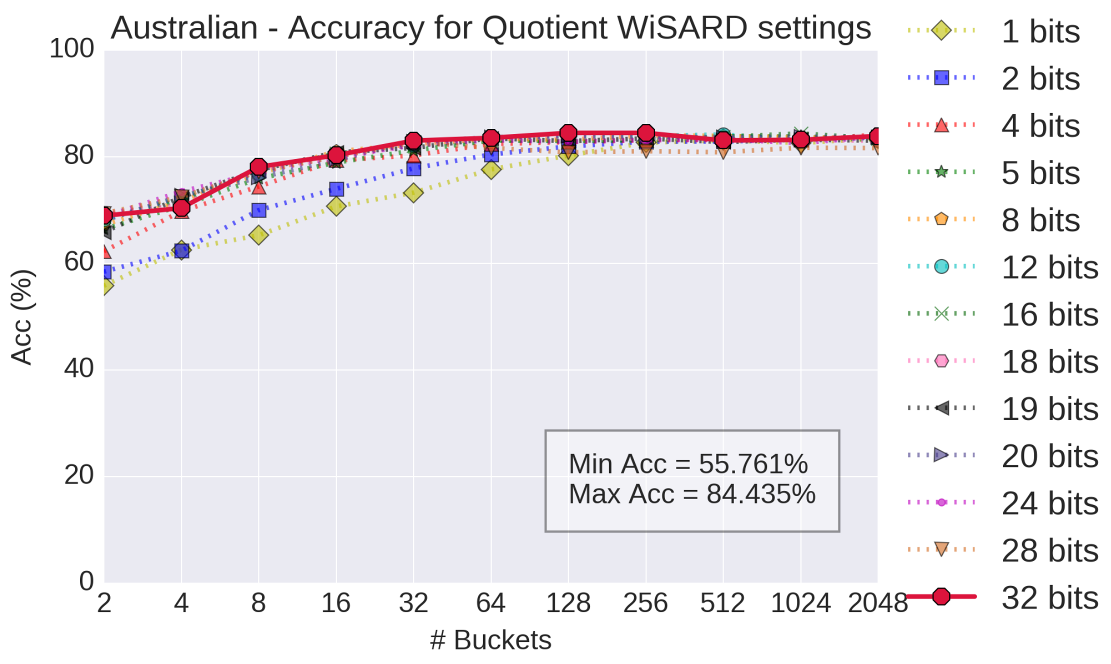


Figure B.4: Accuracy results of Australian dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

B.8 Letter

The results of letter dataset are presented in Figure B.15 and Figure B.16 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

B.9 Liver

The results of liver dataset are presented in Figure B.17 and Figure B.18 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

B.10 MNIST

The results of mnist dataset are presented in Figure B.19 and Figure B.20 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

B.11 Mushroom

The results of mushroom dataset are presented in Figure B.21 and Figure B.22 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

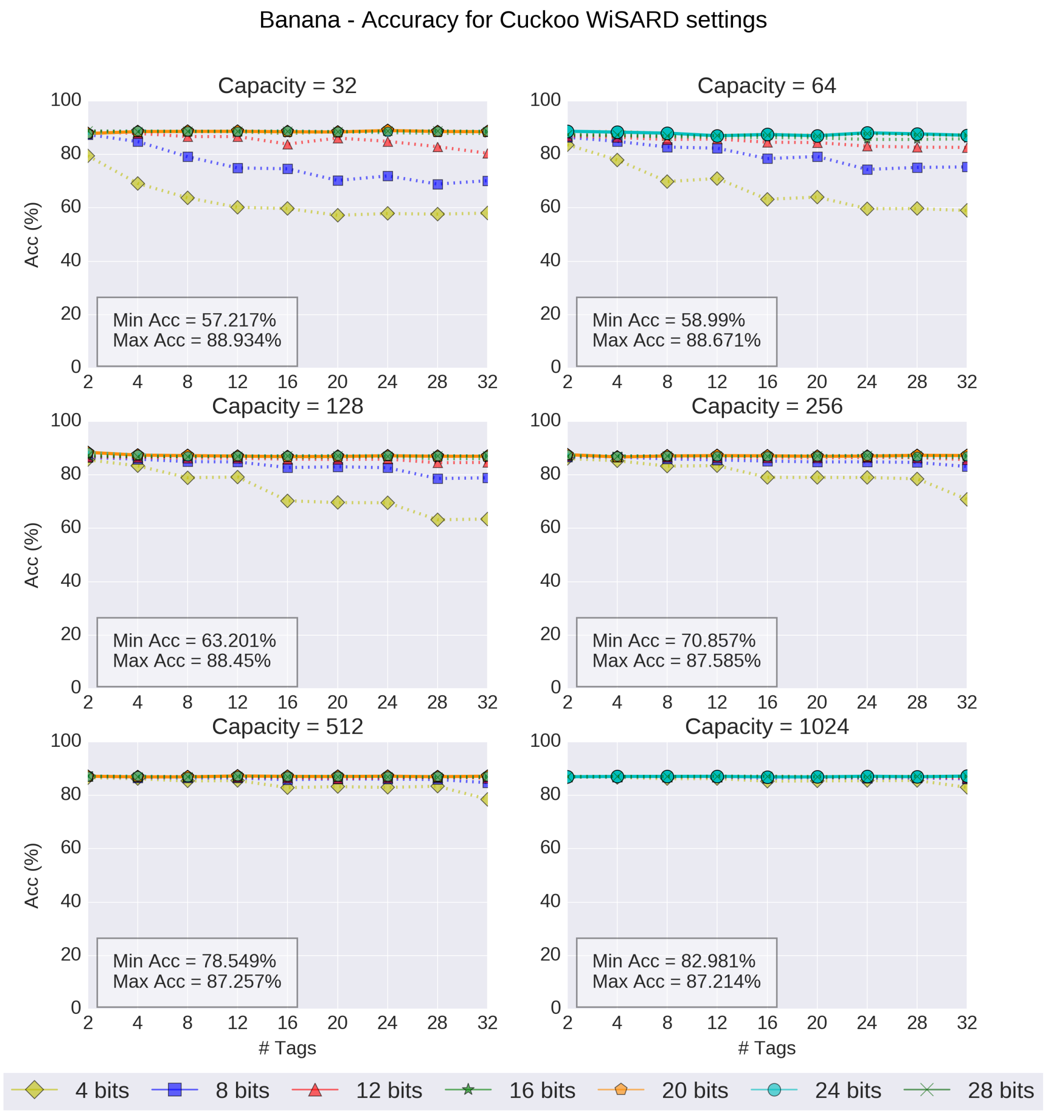


Figure B.5: Accuracy results of Banana dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

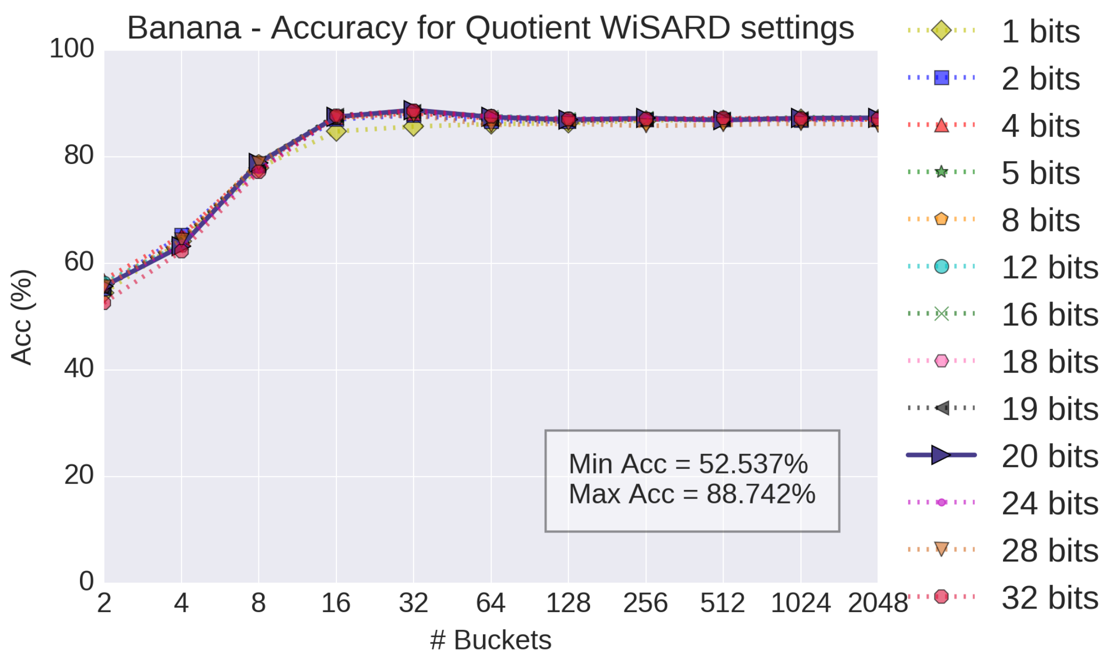


Figure B.6: Accuracy results of Banana dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

B.12 Satimage

The results of satimage dataset are presented in Figure B.23 and Figure B.24 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

B.13 Segment

The results of segment dataset are presented in Figure B.25 and Figure B.26 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

B.14 Shuttle

The results of shuttle dataset are presented in Figure B.27 and Figure B.28 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

B.15 Vehicle

The results of vehicle dataset are presented in Figure B.29 and Figure B.30 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

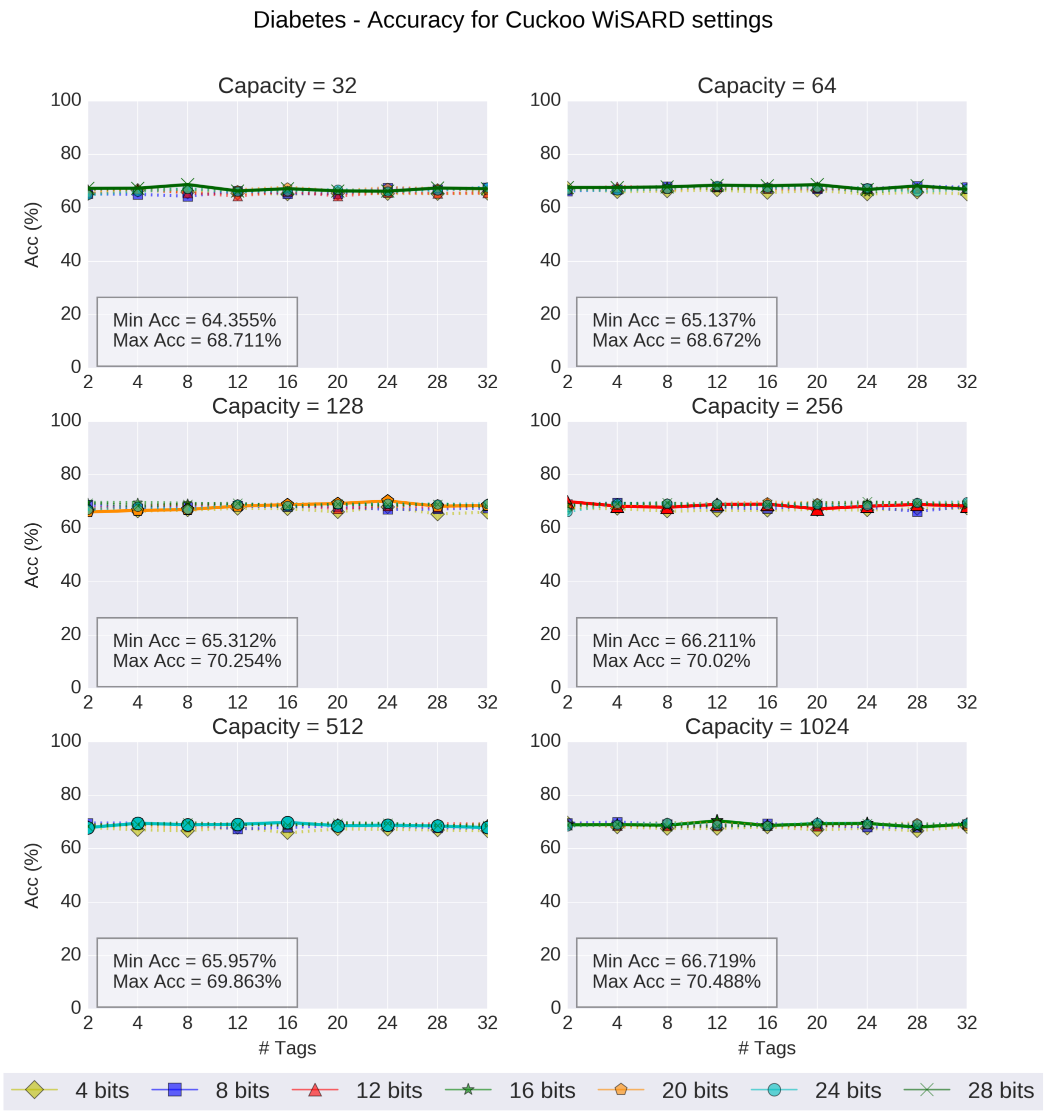


Figure B.7: Accuracy results of Diabetes dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

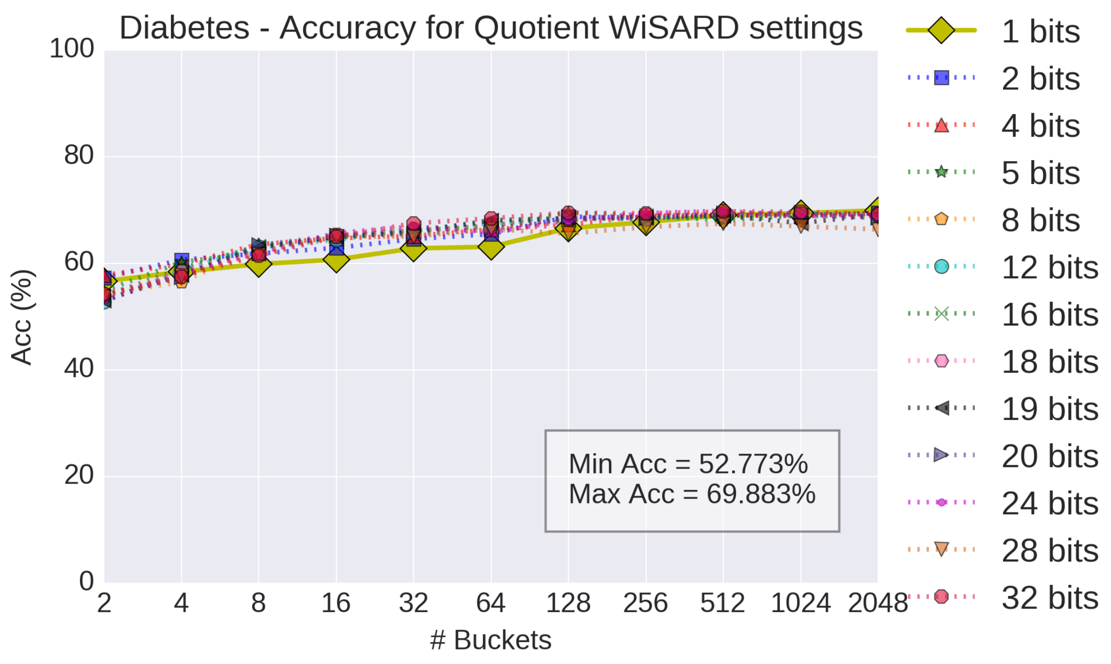


Figure B.8: Accuracy results of Diabetes dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

B.16 Vowel

The results of vowel dataset are presented in Figure B.31 and Figure B.32 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

B.17 Wine

The results of wine dataset are presented in Figure B.33 and Figure B.34 using various configurations of Cuckoo WiSARD and Quotient WiSARD, respectively.

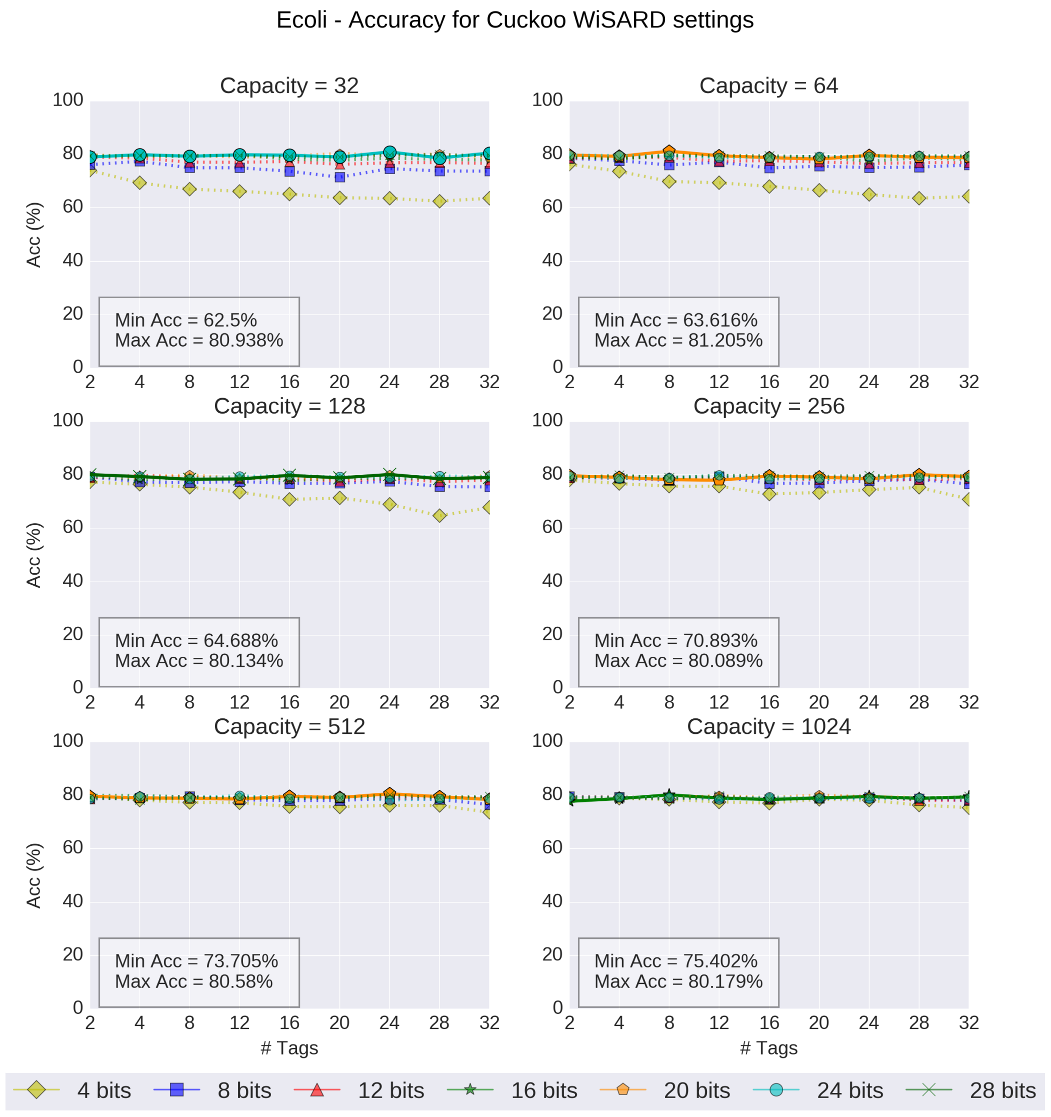


Figure B.9: Accuracy results of Ecoli dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

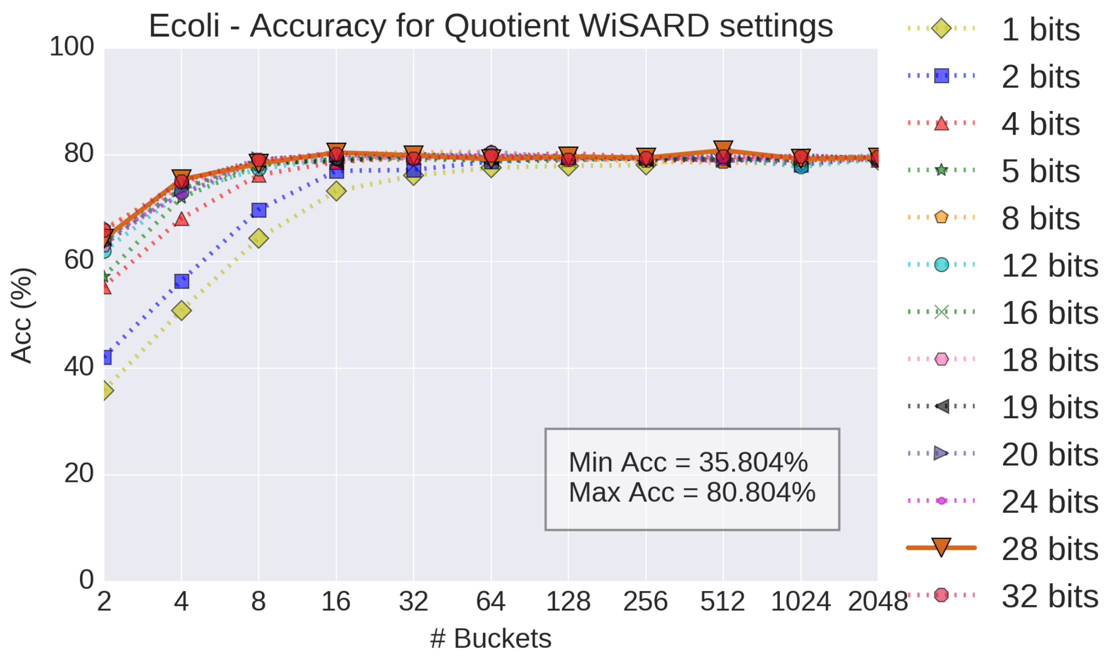


Figure B.10: Accuracy results of Ecoli dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

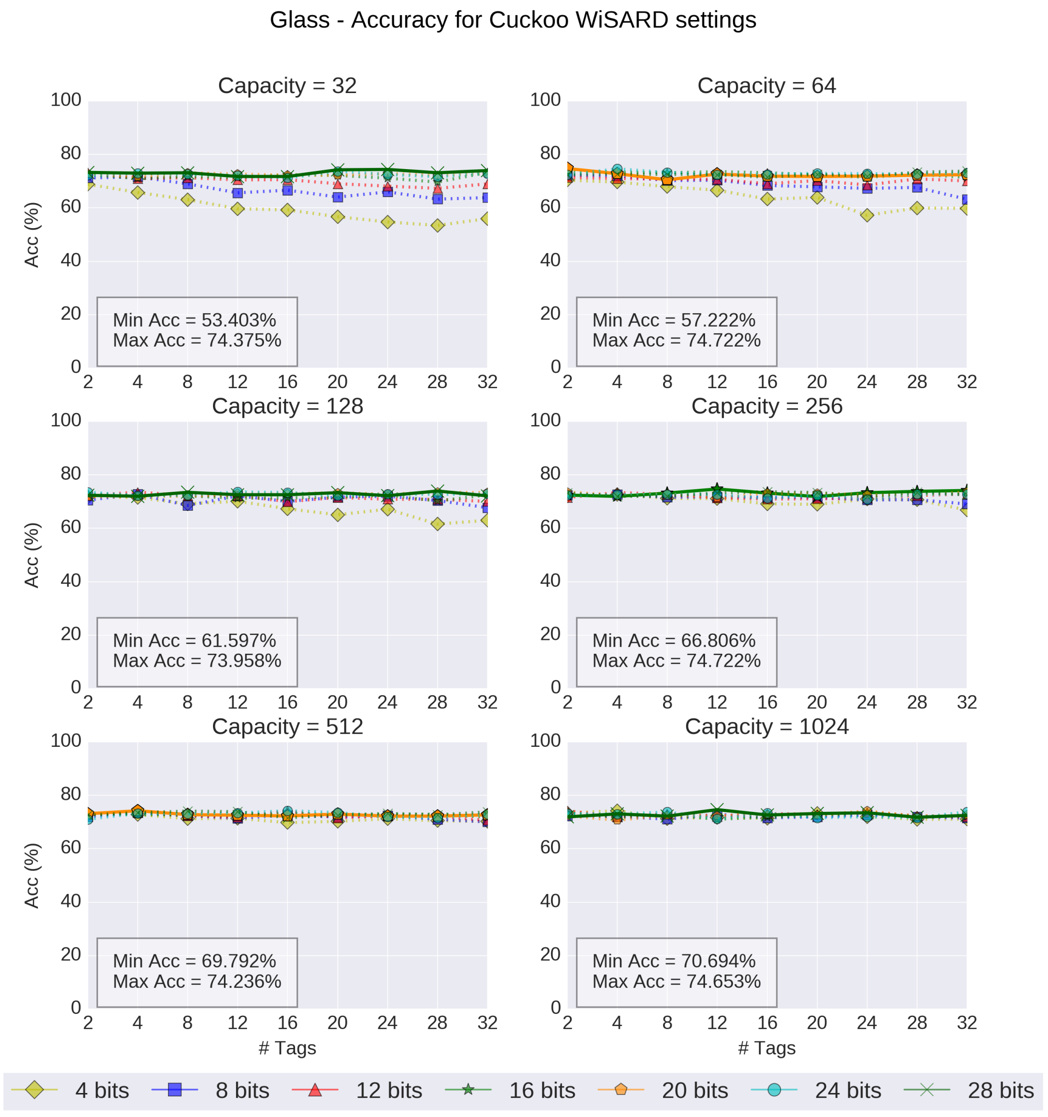


Figure B.11: Accuracy results of Glass dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

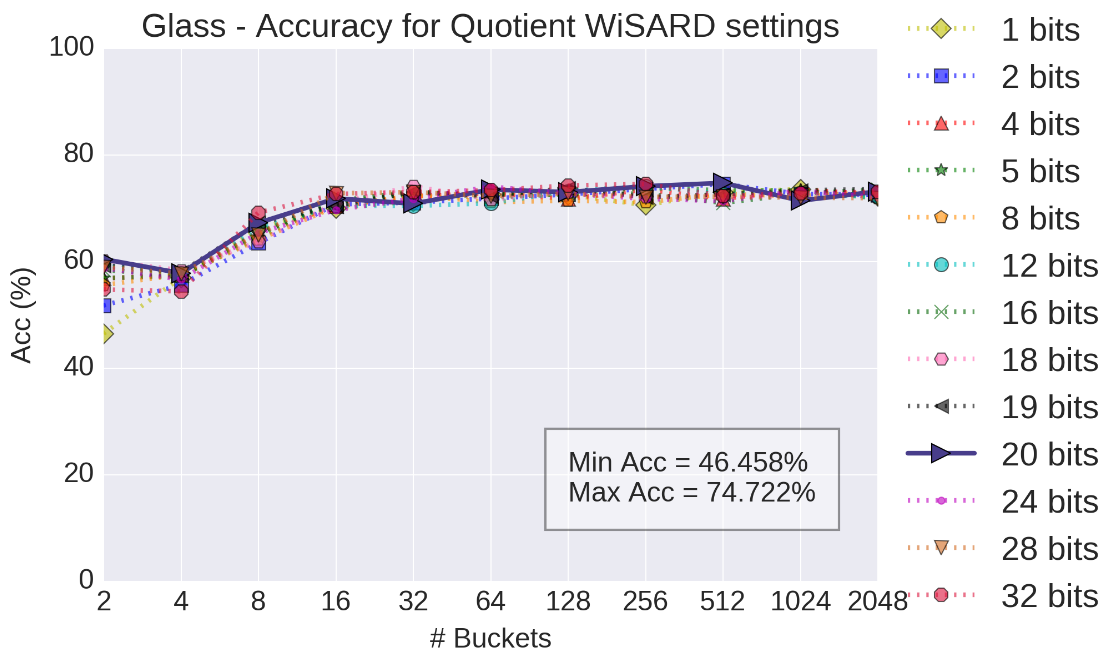


Figure B.12: Accuracy results of Glass dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

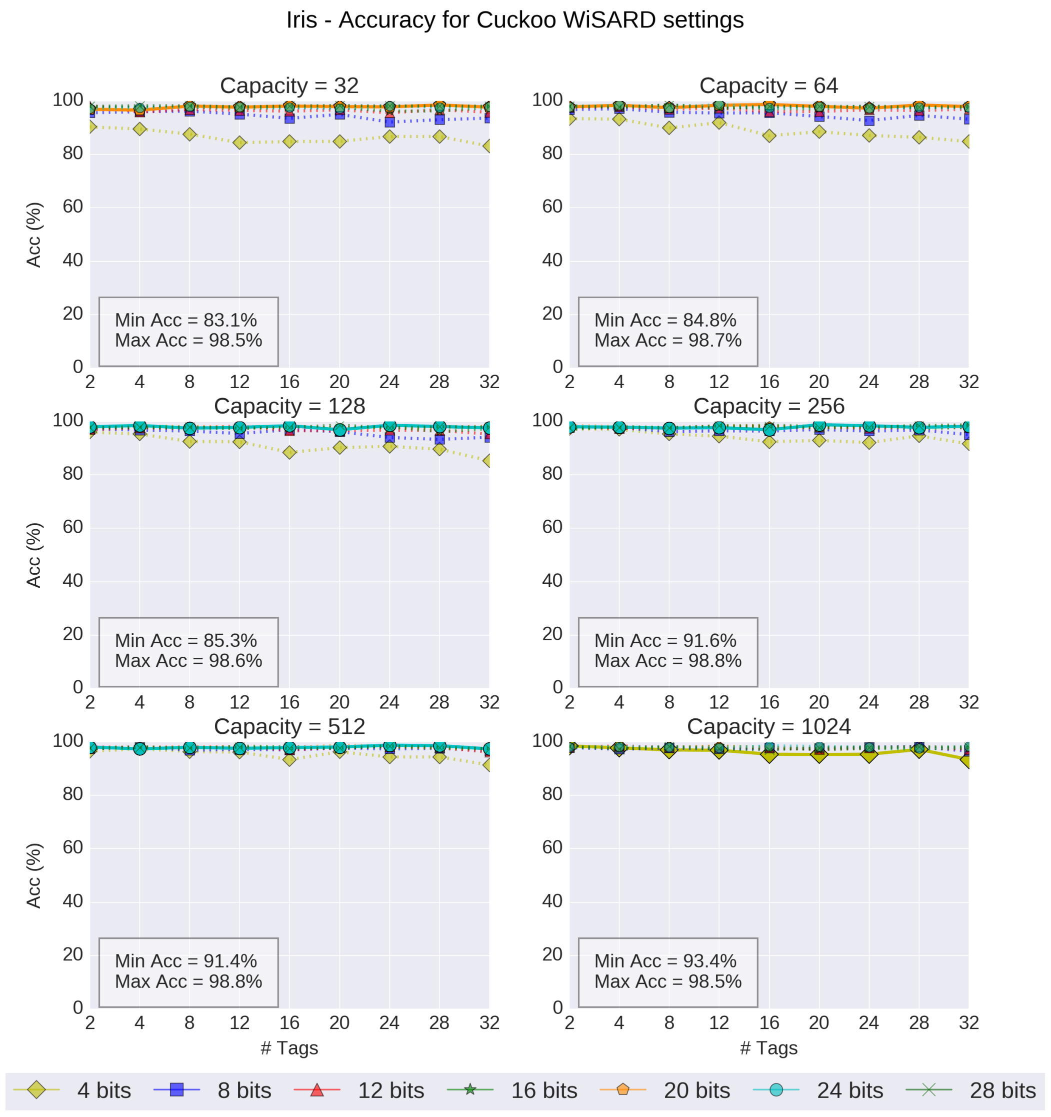


Figure B.13: Accuracy results of Iris dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

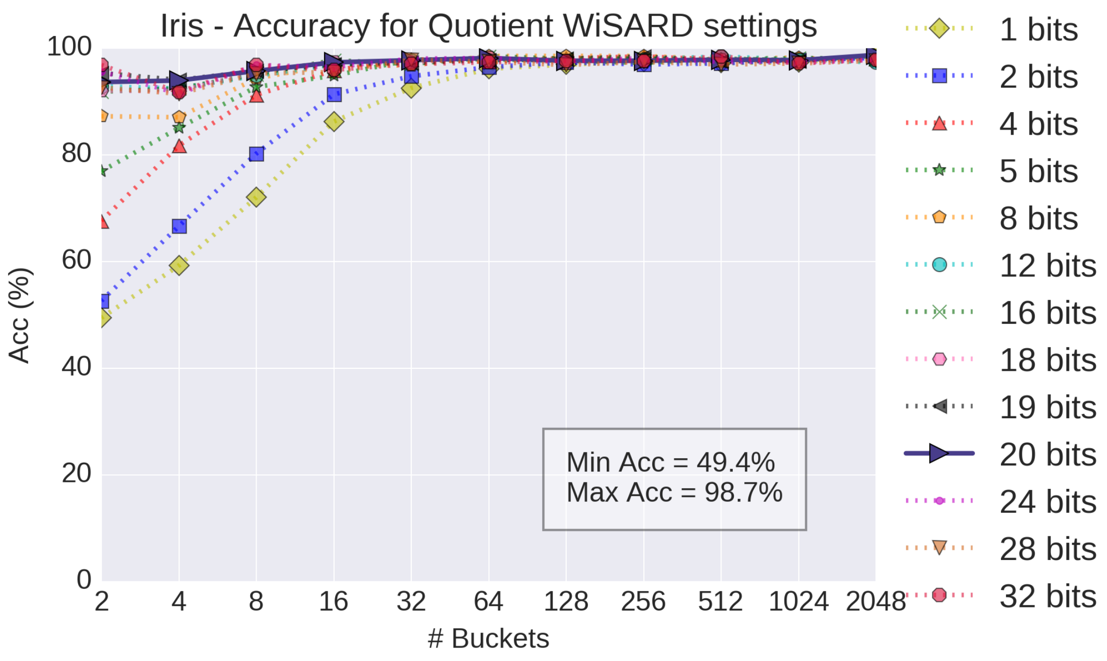


Figure B.14: Accuracy results of Iris dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

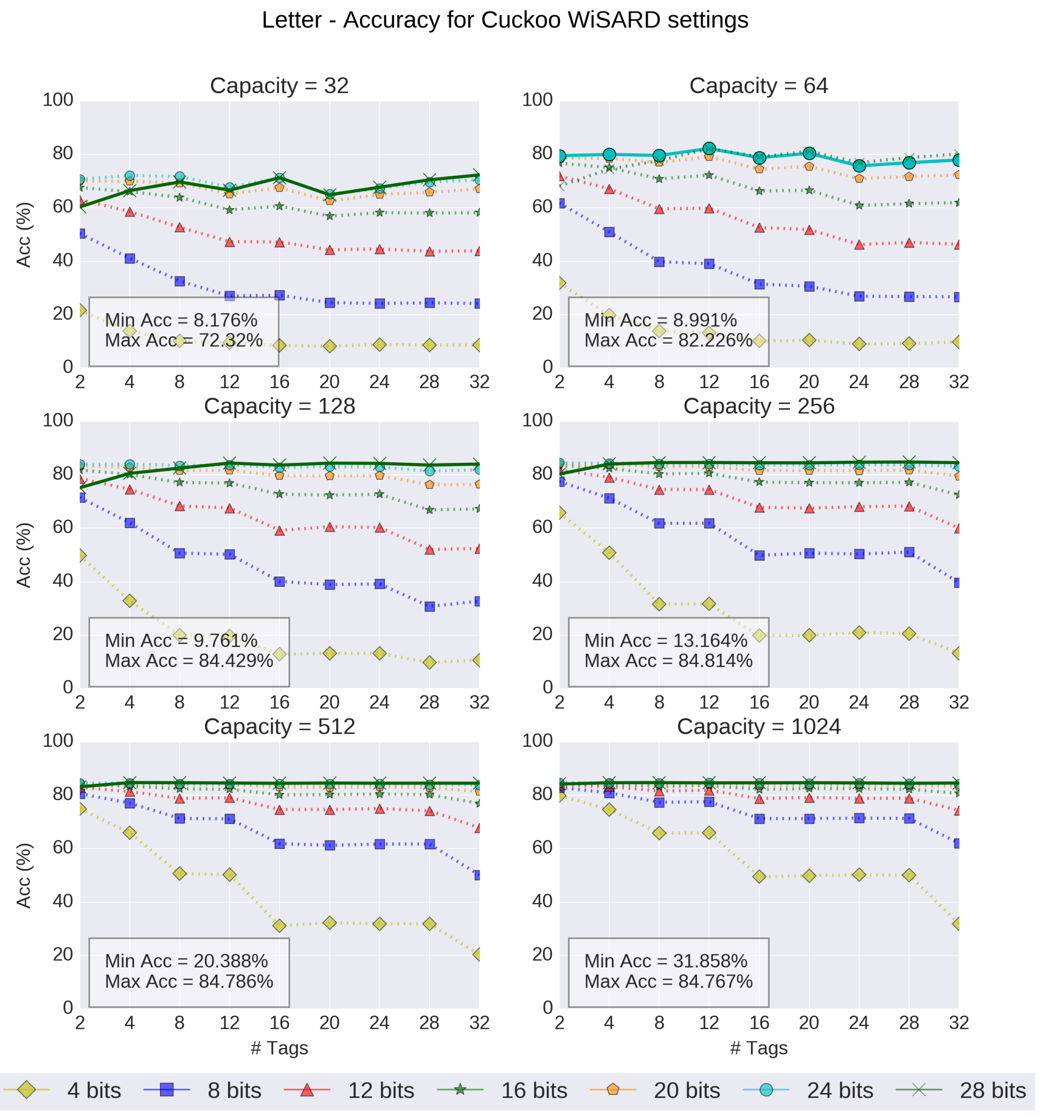


Figure B.15: Accuracy results of Letter dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

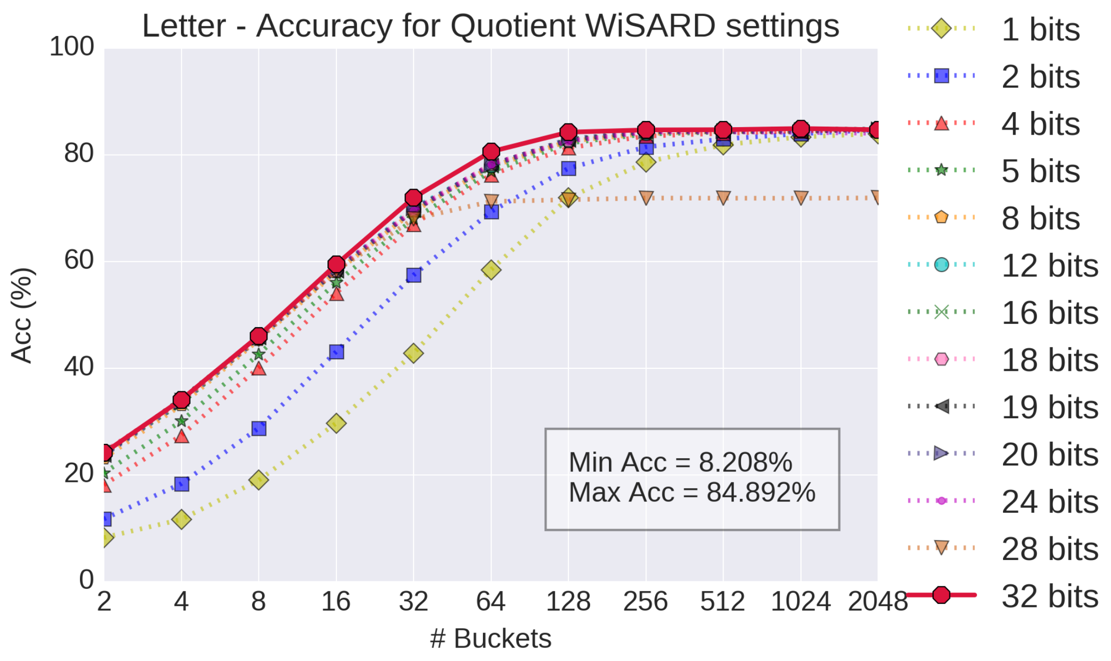


Figure B.16: Accuracy results of Letter dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

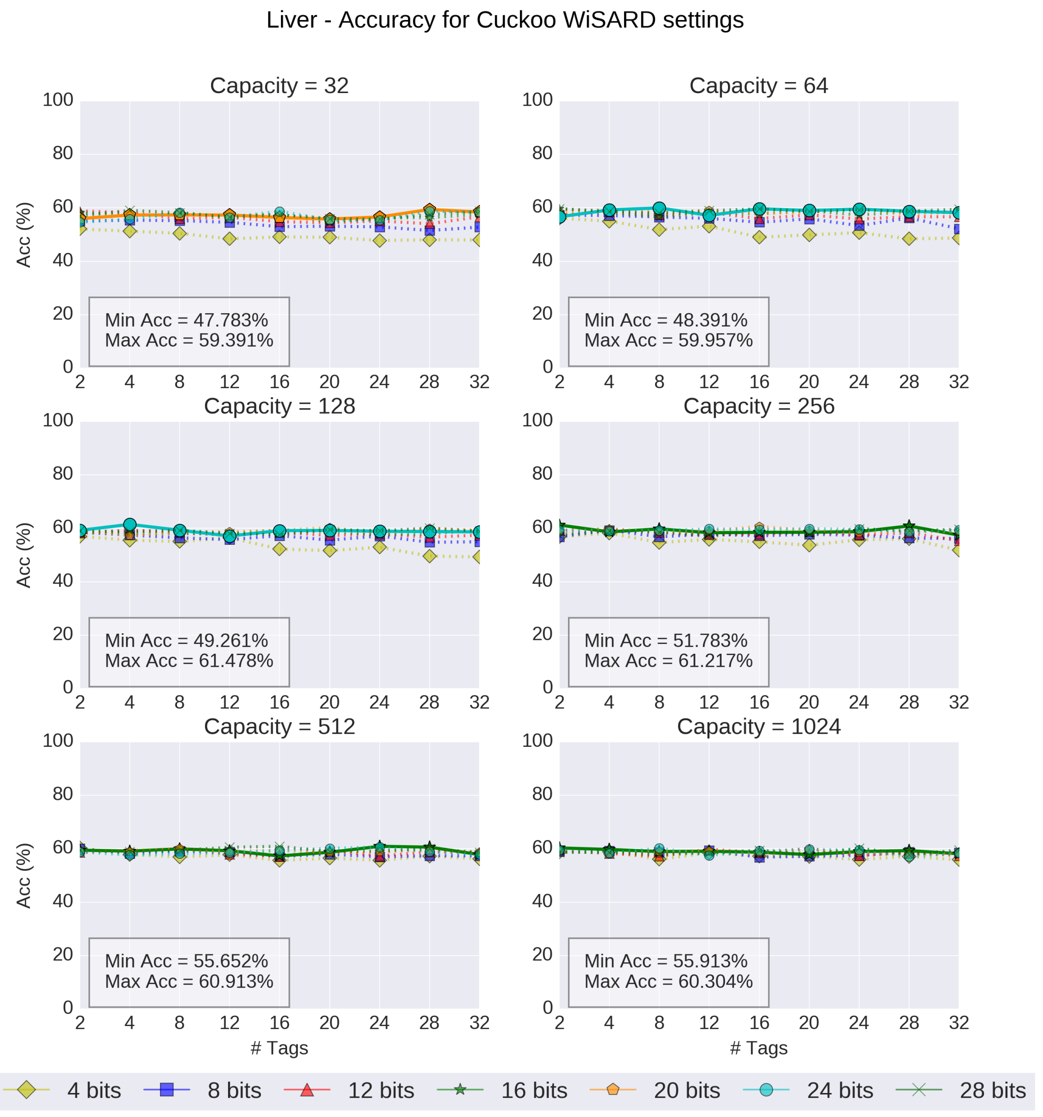


Figure B.17: Accuracy results of Liver dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

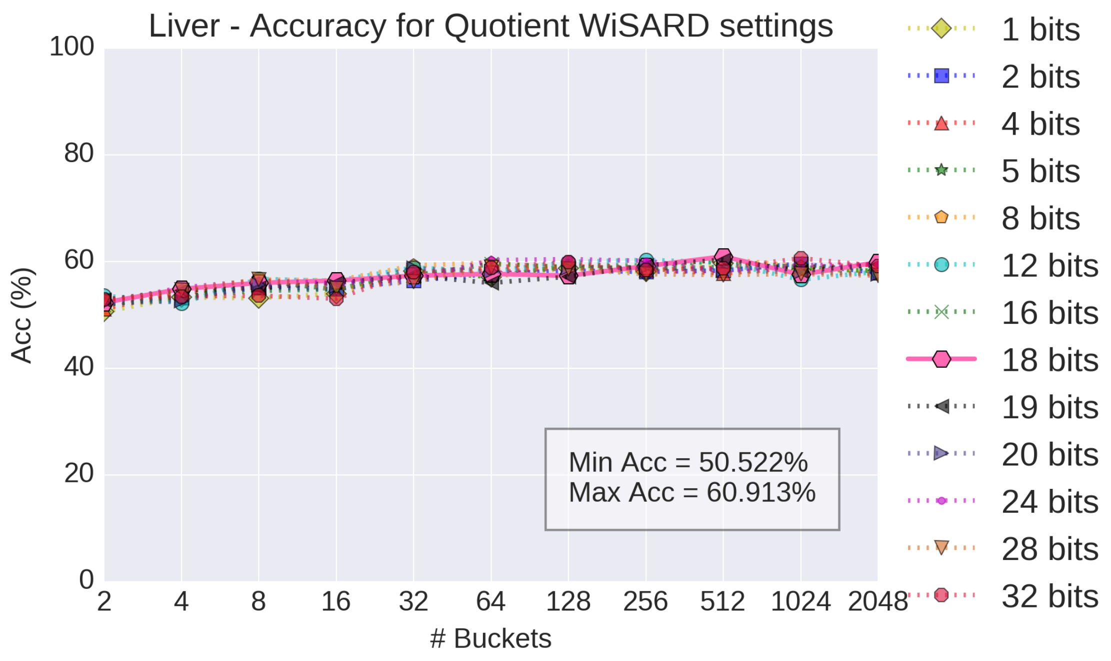


Figure B.18: Accuracy results of Liver dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

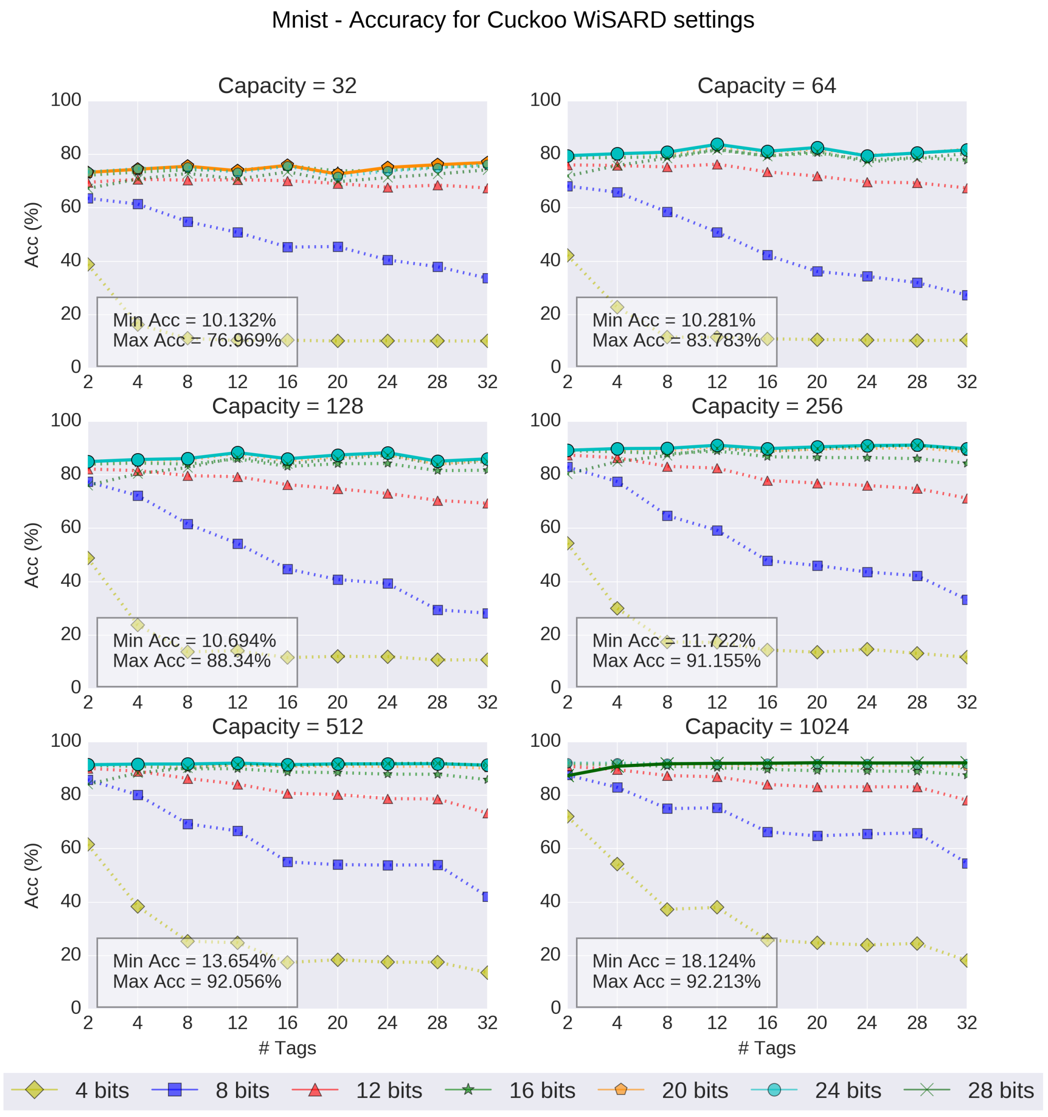


Figure B.19: Accuracy results of MNIST dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

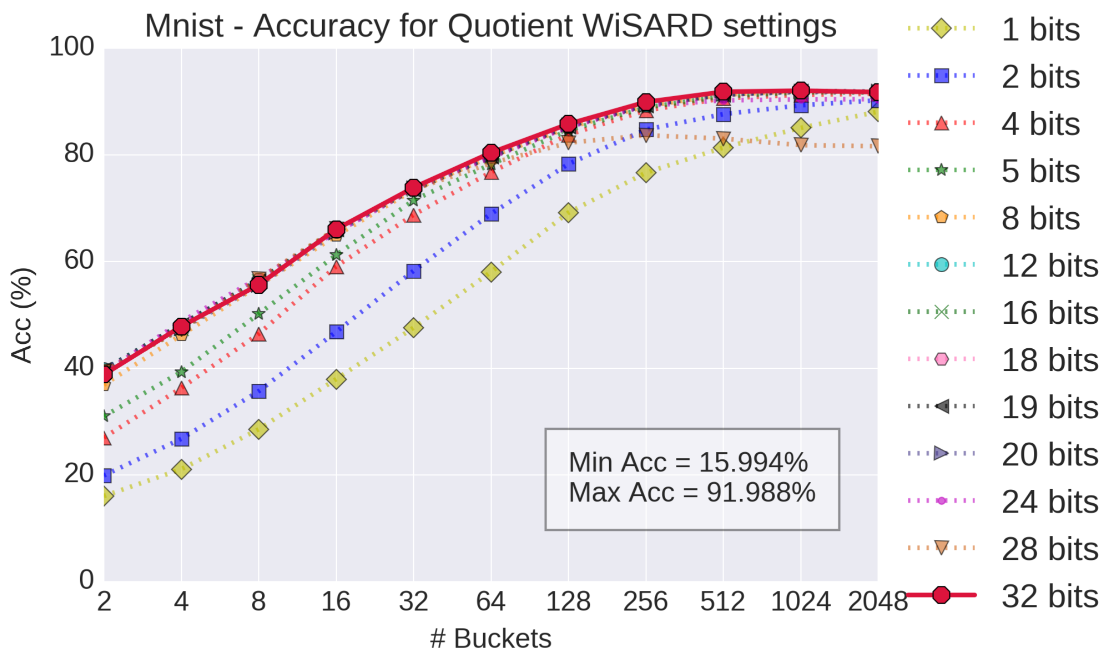


Figure B.20: Accuracy results of MNIST dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

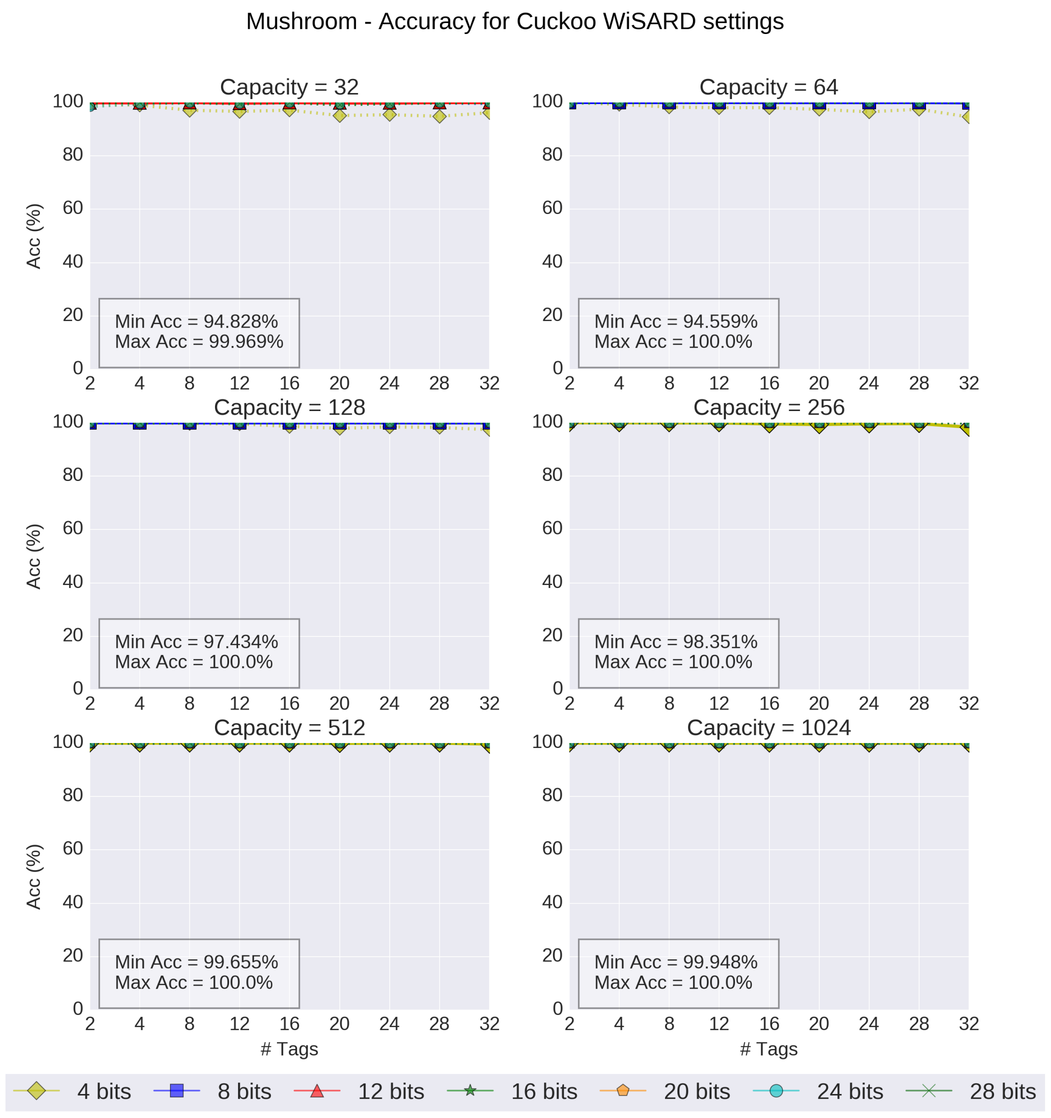


Figure B.21: Accuracy results of Mushroom dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

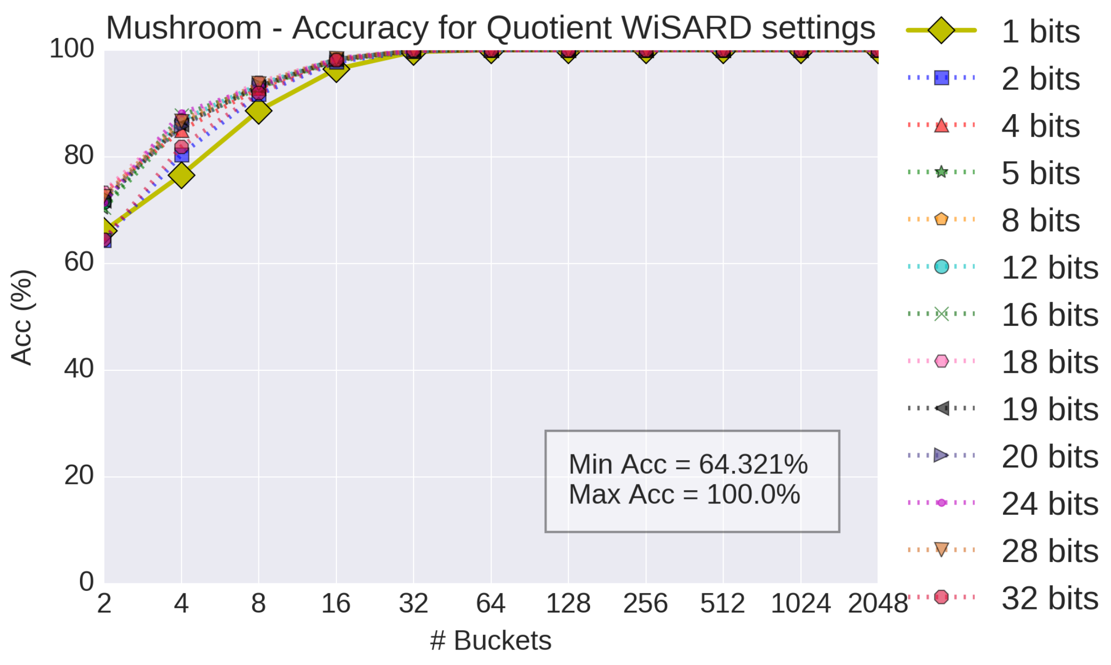


Figure B.22: Accuracy results of Mushroom dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

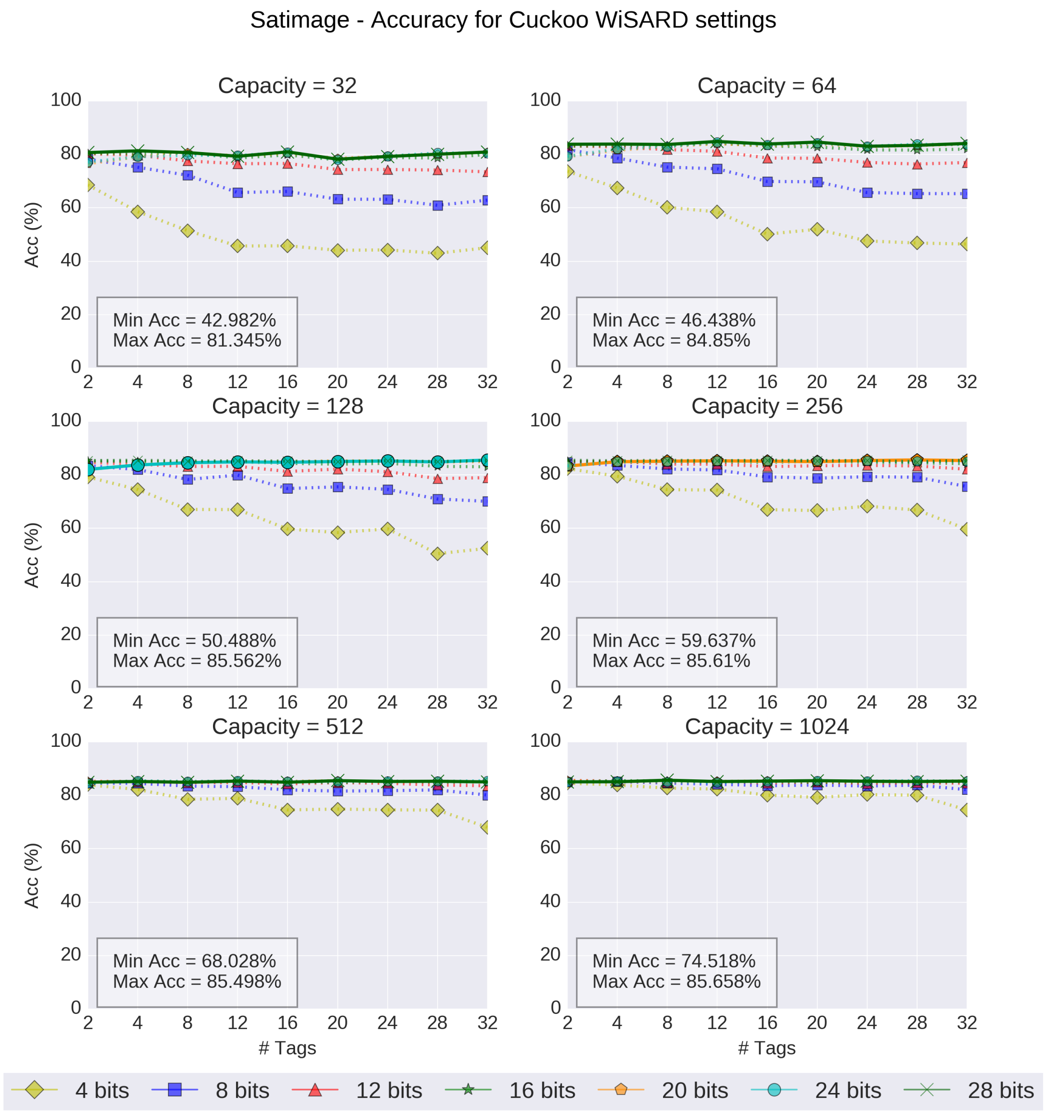


Figure B.23: Accuracy results of Satimage dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

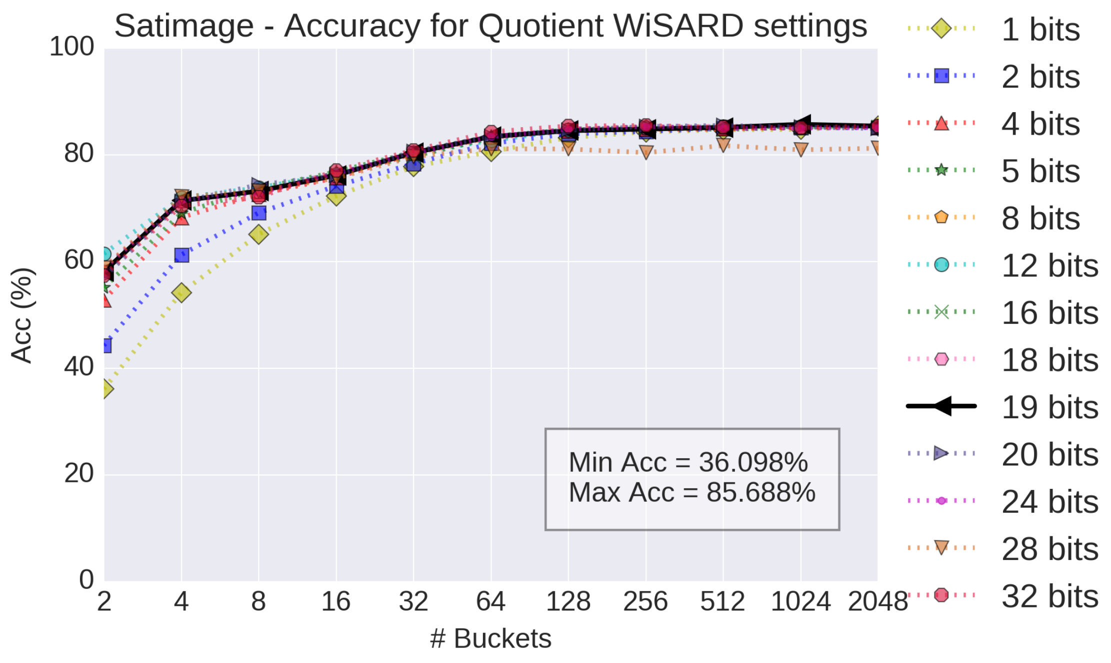


Figure B.24: Accuracy results of Satimage dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

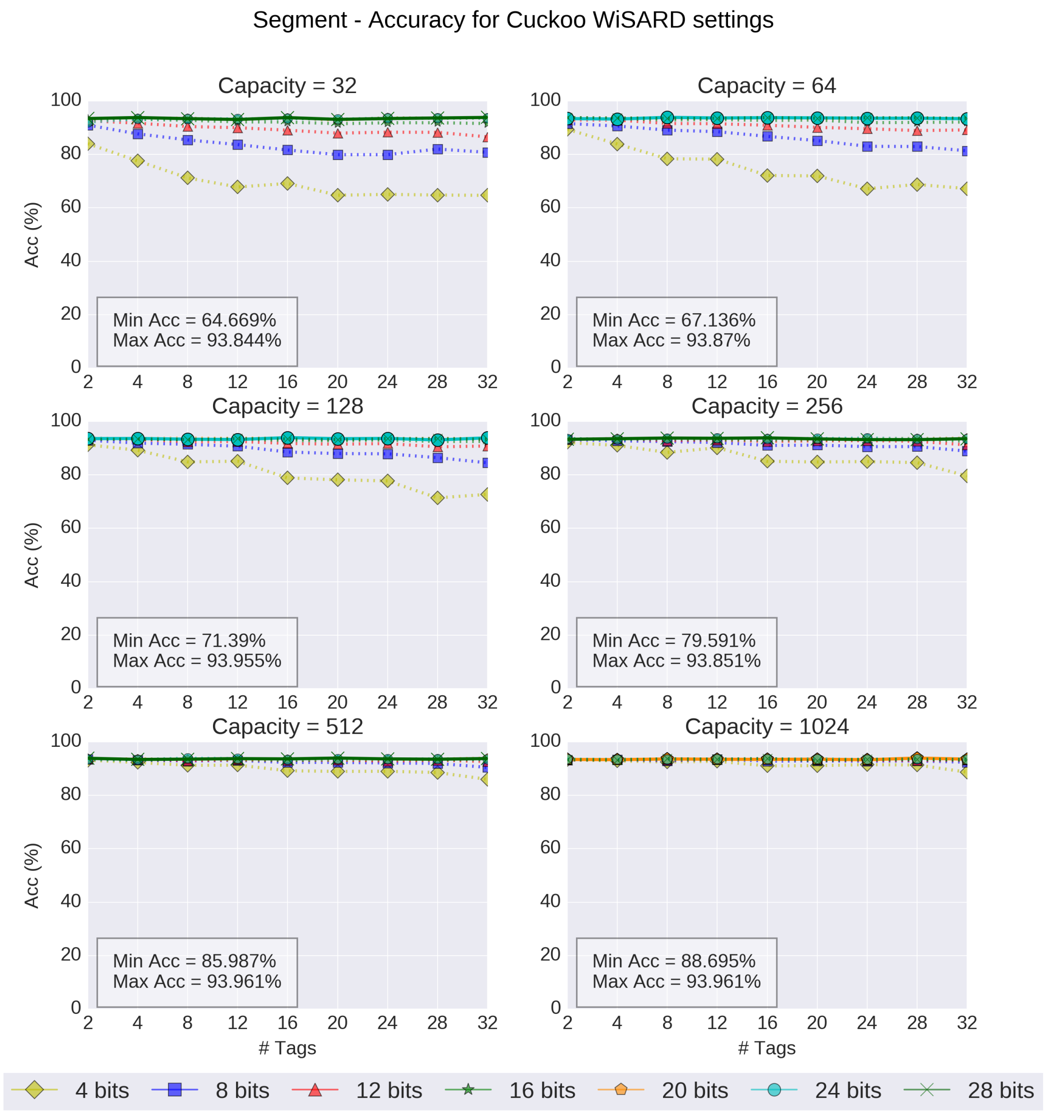


Figure B.25: Accuracy results of Segment dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

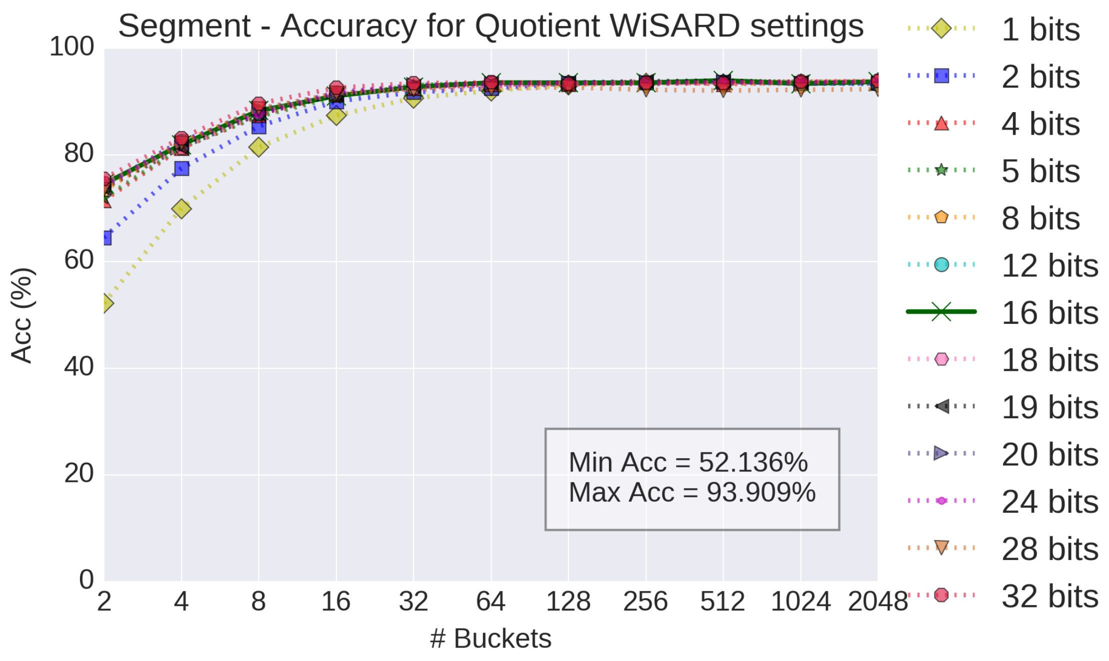


Figure B.26: Accuracy results of Segment dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

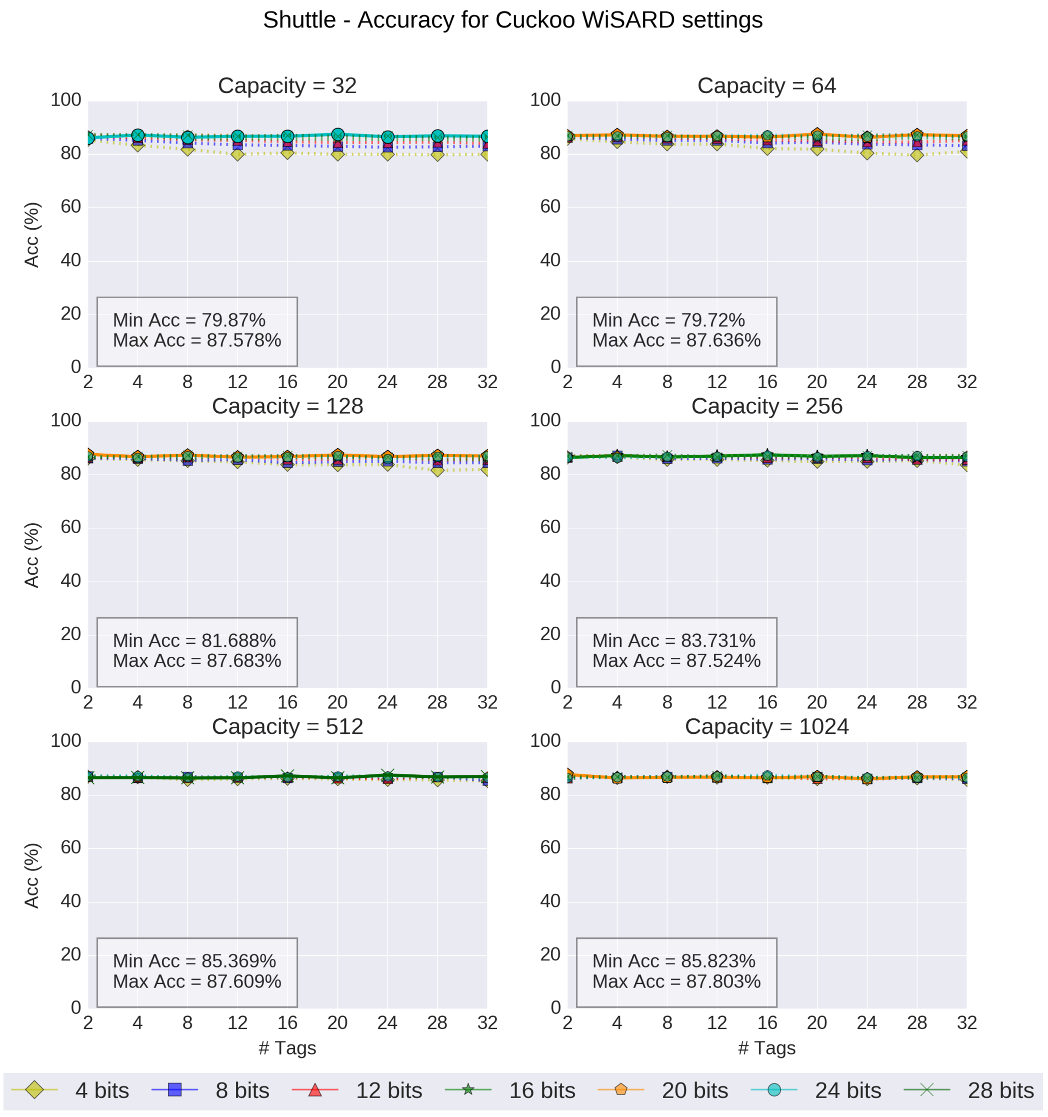


Figure B.27: Accuracy results of Shuttle dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

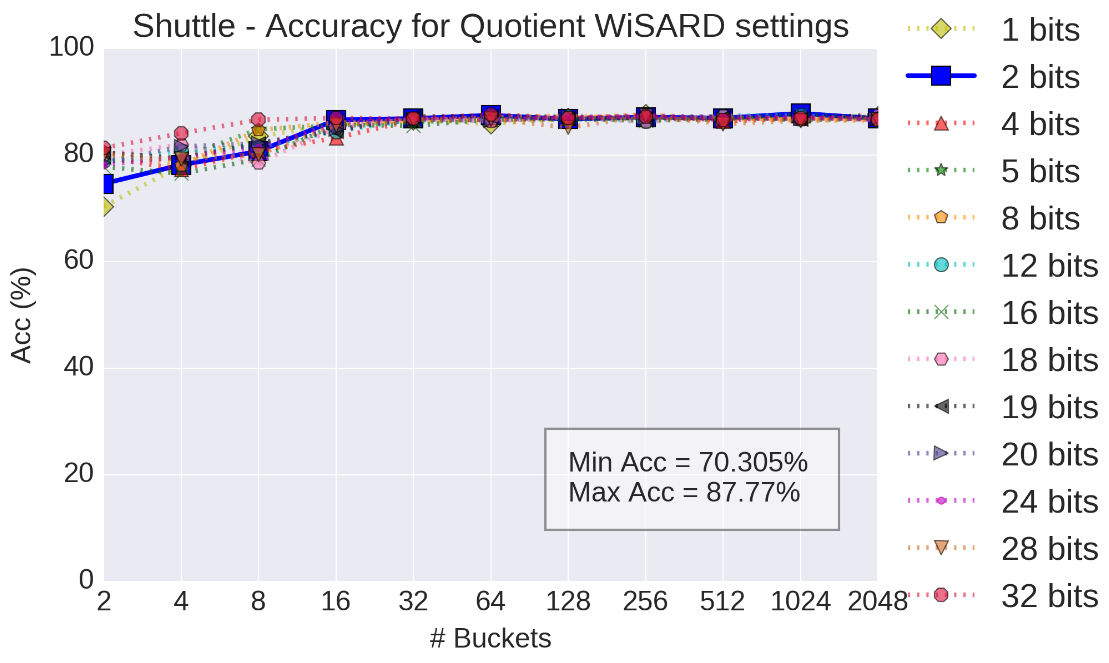


Figure B.28: Accuracy results of Shuttle dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

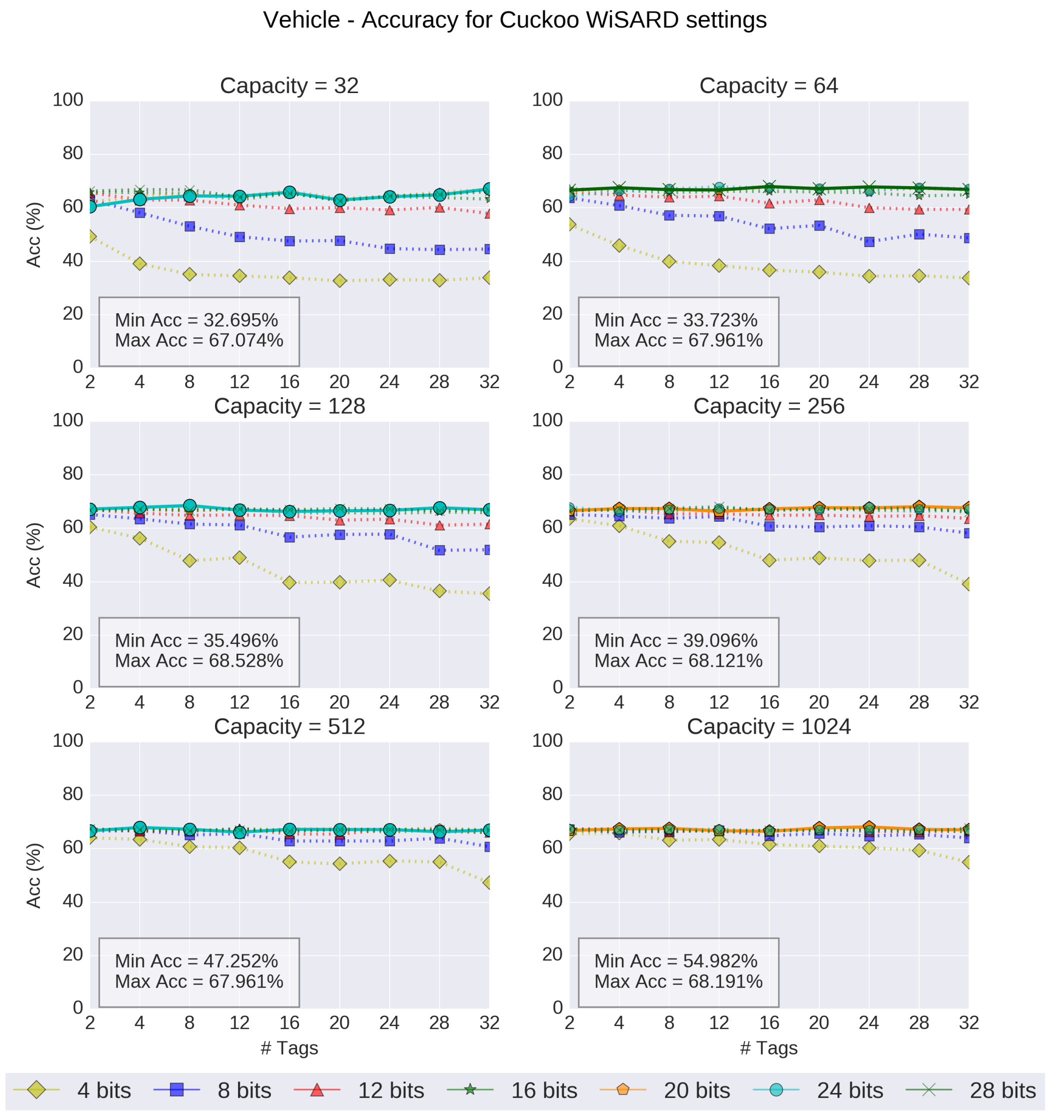


Figure B.29: Accuracy results of Vehicle dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

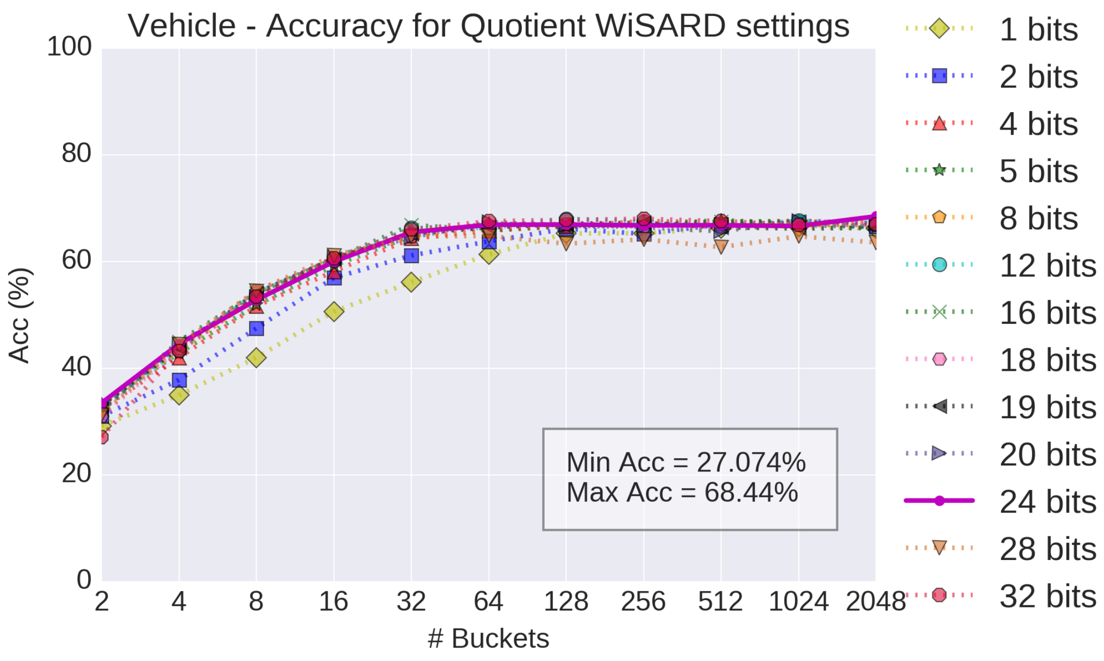


Figure B.30: Accuracy results of Vehicle dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

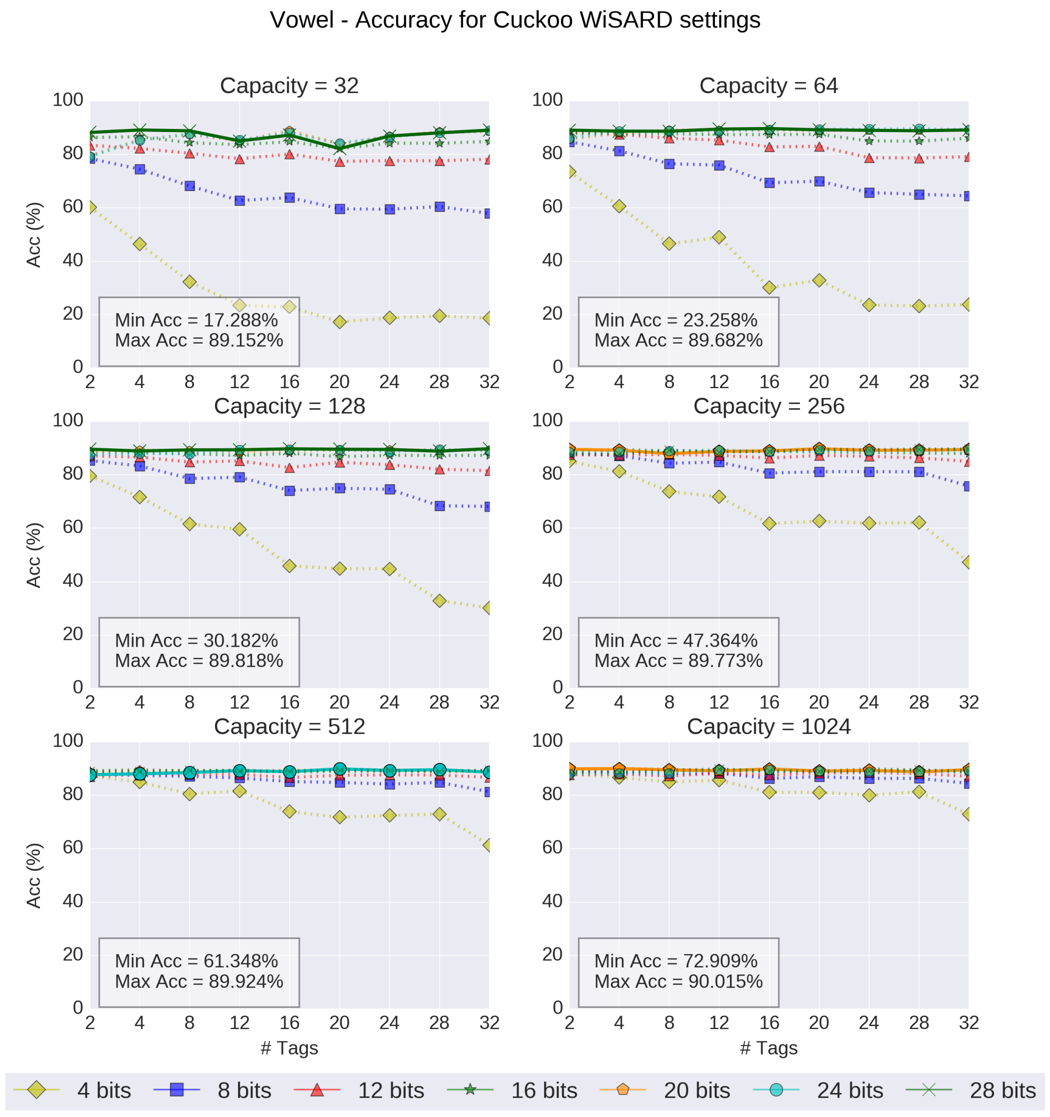


Figure B.31: Accuracy results of Vowel dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

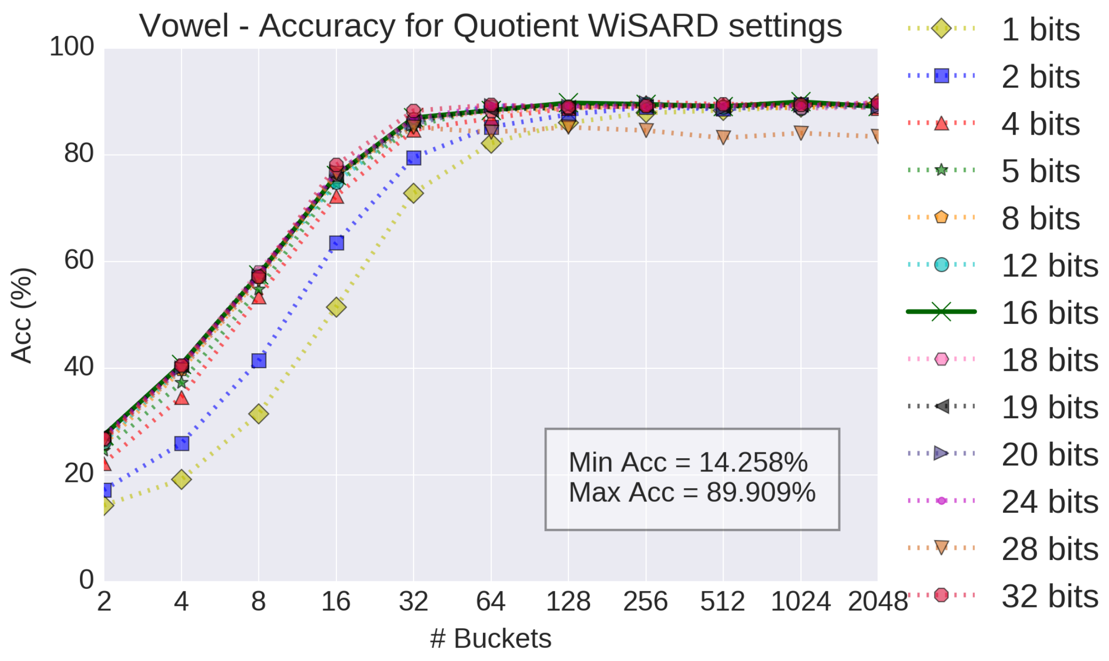


Figure B.32: Accuracy results of Vowel dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

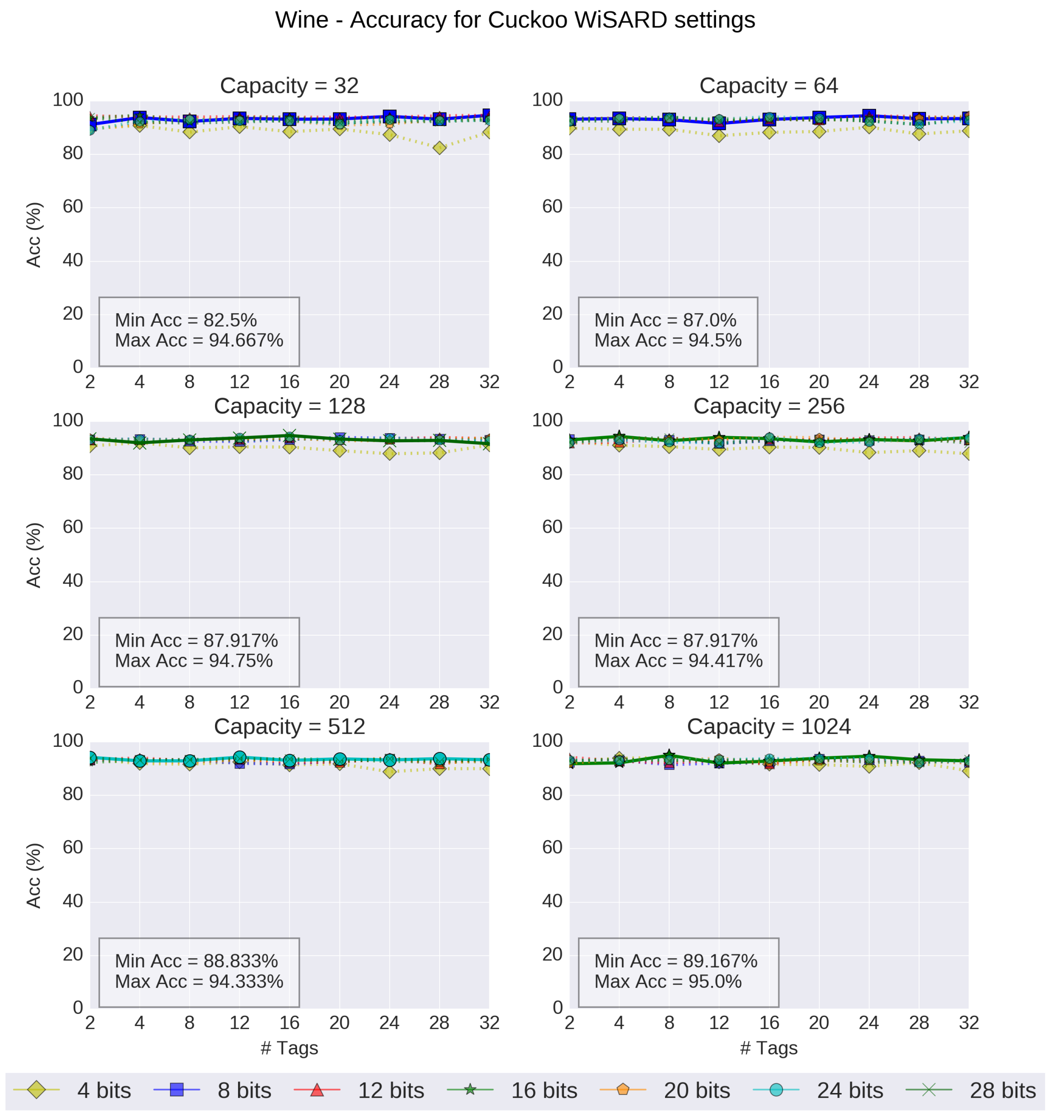


Figure B.33: Accuracy results of Wine dataset when varying capacity, number of tags (entries) per bucket and tag bits of Cuckoo WiSARD. The x axis represents the number of tags per bucket and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

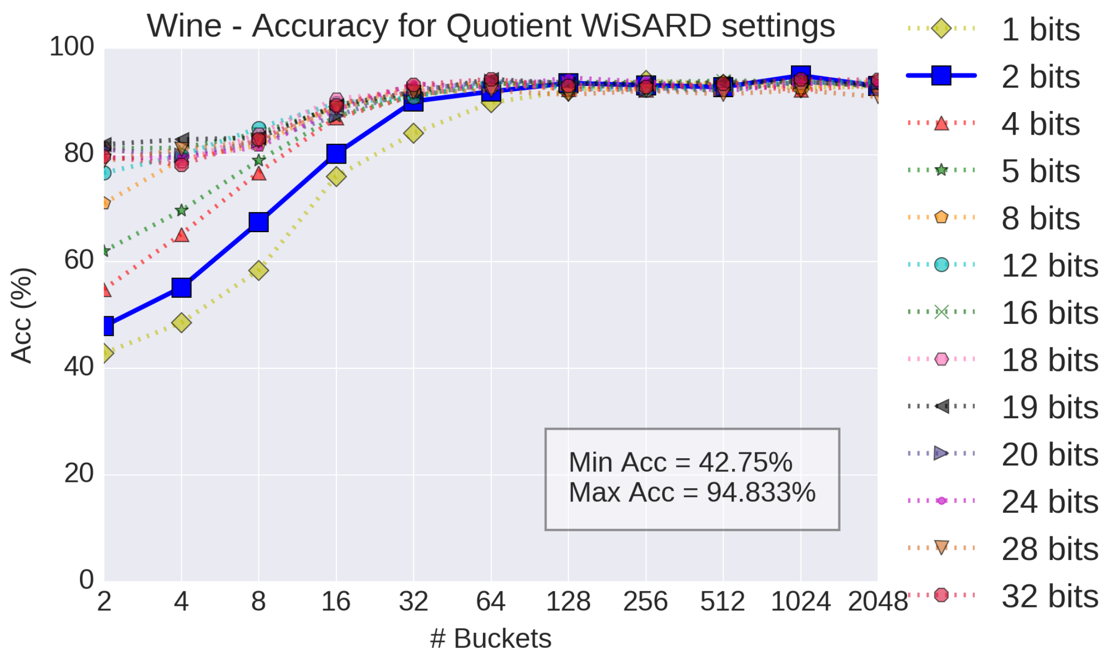


Figure B.34: Accuracy results of Wine dataset when varying number of buckets (quotient bits) and tag bits (remainder bits) of Quotient WiSARD. The x axis represents the number of buckets and each line represents one tag bit setting. The highlighted line indicates the best result in term of accuracy.

Appendix C

List of Publications

C.1 Journal Articles

1. Leandro Santiago; VERONA, Leticia; RANGEL, Fábio; FARIA, F. F.; MENASCHE, Daniel S.; et al. “Weightless Neural Networks as Memory Segmented Bloom Filters”, Neurocomputing 2019. **Submitted**.

2. Leandro Santiago; PATIL, VINAY C.; PRADO, CHARLES B.; ALVES, TIAGO A. O.; MARZULO, LEANDRO A.J.; FRANCA, FELIPE M.G.; KUNDU, SANDIP. “Design of Robust, High-Entropy Strong PUFs via Weightless Neural Network”, Journal of Hardware and Systems Security. , v.3, p.1 - 15, 2019. [3]

C.2 In Conference Proceedings

1. Santiago, Leandro; PATIL, VINAY C.; MARZULO, LEANDRO A.J.; FRANCA, FELIPE M.G.; KUNDU, SANDIP. “Efficient Testing of Physically Unclonable Functions for Uniqueness”, The 28th IEEE Asian Test Symposium (ATS 2019), India. **Submitted**.

2. Santiago, Leandro; Marzulo, Leandro A. J.; ALVES, TIAGO A. O.; França, Felipe M. G.; Koren, Israel; Kundu, Sandip. “Building a Portable Deeply-Nested Implicit Information Flow Tracking”, The 37th IEEE International Conference on Computer Design, Abu Dhabi. **Submitted**.

3. Santiago, Leandro; Ferreira, Victor C. ; Goldstein, Brunno F. ; Nery, Alexandre S. ; Marzulo, Leandro A. J. ; Kundu, Sandip; França, Felipe M. G. . “Hardware-Accelerated Similarity Search with Multi-Index Hashing”, The 17th IEEE International Conference on Pervasive Intelligence and Computing (PICom 2019), Japan.

4. Leandro Santiago; VERONA, Leticia; RANGEL, Fábio; FARIA, F. F.; MENASCHE, Daniel S.; et al. “Memory Efficient Weightless Neural Network using Bloom Filter”, In: ESANN - European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, 2019, Belgium. ESANN 2019 proceedings. , 2019. [110]

5. SANTIAGO, LEANDRO; PATIL, VINAY C.; PRADO, CHARLES B.; ALVES, TIAGO A. O.; MARZULO, LEANDRO A.J.; FRANCA, FELIPE M.G.; KUNDU, SANDIP. “Realizing strong PUF from weak PUF via neural computing”, In: IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2017, Cambridge. 2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT). IEEE, 2017. p.1 - 6. [2]

Bibliography

[1] COPPOLINO, L., D'ANTONIO, S., MAZZEO, G., et al. "A comprehensive survey of hardware-assisted security: From the edge to the cloud", *Internet of Things*, v. 6, pp. 100055, 2019. ISSN: 2542-6605. doi: https://doi.org/10.1016/j.iot.2019.100055. Disponível em: <http://www.sciencedirect.com/science/article/pii/S2542660519300101>.

[2] SANTIAGO, L., PATIL, V. C., PRADO, C. B., et al. "Realizing strong PUF from weak PUF via neural computing". In: *2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, Oct 2017. doi: 10.1109/DFT.2017.8244433.

[3] SANTIAGO DE ARAÚJO, L., C. PATIL, V., B. PRADO, C., et al. "Design of Robust, High-Entropy Strong PUFs via Weightless Neural Network", *Journal of Hardware and Systems Security*, May 2019. ISSN: 2509-3436. doi: 10.1007/s41635-019-00071-z. Disponível em: <https://doi.org/10.1007/s41635-019-00071-z>.

[4] DALALAH, A., BABA, S., TUBAISHAT, A. "New Hardware Architecture for Bit-counting". In: *Proceedings of the 5th WSEAS International Conference on Applied Computer Science*, ACOS'06, pp. 118–128, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS). ISBN: 960-8457-43-2. Disponível em: <http://dl.acm.org/citation.cfm?id=1973598.1973623>.

[5] RAKOTONDRAVONY, N., TAUBMANN, B., MANDARAWI, W., et al. "Classifying malware attacks in IaaS cloud environments", *Journal of Cloud Computing*, v. 6, n. 1, pp. 26, Dec 2017. ISSN: 2192-113X. doi: 10.1186/s13677-017-0098-8. Disponível em: <https://doi.org/10.1186/s13677-017-0098-8>.

[6] WU, Z., XU, Z., WANG, H. "Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud". In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security*

12), pp. 159–173, Bellevue, WA, 2012. USENIX. ISBN: 978-931971-95-9. Disponível em: <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/wu>.

[7] STEWIN, P., BYSTROV, I. "Understanding DMA Malware". In: Flegel, U., Markatos, E., Robertson, W. (Eds.), *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 21–41, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN: 978-3-642-37300-8.

[8] ROEMER, R., BUCHANAN, E., SHACHAM, H., et al. "Return-Oriented Programming: Systems, Languages, and Applications", *ACM Trans. Inf. Syst. Secur.*, v. 15, n. 1, pp. 2:1–2:34, mar. 2012. ISSN: 1094-9224. doi: 10.1145/2133375.2133377. Disponível em: <http://doi.acm.org/10.1145/2133375.2133377>.

[9] BLETSCH, T., JIANG, X., FREEH, V. W., et al. "Jump-oriented Programming: A New Class of Code-reuse Attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pp. 30–40, New York, NY, USA, 2011. ACM. ISBN: 978-1-4503-0564-8. doi: 10.1145/1966913.1966919. Disponível em: <http://doi.acm.org/10.1145/1966913.1966919>.

[10] COWAN, C., WAGLE, P., PU, C., et al. "Buffer overflows: attacks and defenses for the vulnerability of the decade". In: *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, pp. 227–237, Dec 2003. doi: 10.1109/FITS.2003.1264935.

[11] KELSEY, J., SCHNEIER, B., WAGNER, D., et al. "Cryptanalytic Attacks on Pseudorandom Number Generators". In: Vaudenay, S. (Ed.), *Fast Software Encryption*, pp. 168–188, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN: 978-3-540-69710-7.

[12] FRANCILLON, A., CASTELLUCCIA, C. "Code Injection Attacks on Harvard-architecture Devices". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pp. 15–26, New York, NY, USA, 2008. ACM. ISBN: 978-1-59593-810-7. doi: 10.1145/1455770.1455775. Disponível em: <http://doi.acm.org/10.1145/1455770.1455775>.

[13] CHECKOWAY, S., SHACHAM, H. "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface", *SIGPLAN Not.*, v. 48, n. 4, pp. 253–264, mar. 2013. ISSN: 0362-1340. doi: 10.1145/2499368.2451145. Disponível em: <http://doi.acm.org/10.1145/2499368.2451145>.

[14] ANDERSON, R. J. *Security Engineering: A Guide to Building Dependable Distributed Systems*. 1st ed. New York, NY, USA, John Wiley & Sons, Inc., 2001. ISBN: 0471389226.

[15] GASSEND, B., CLARKE, D., VAN DIJK, M., et al. "Silicon Physical Random Functions". In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pp. 148–160, New York, NY, USA, 2002. ACM. ISBN: 1-58113-612-9. doi: 10.1145/586110.586132. Disponível em: <http://doi.acm.org/10.1145/586110.586132>.

[16] PAPPU, R., RECHT, B., TAYLOR, J., et al. "Physical One-Way Functions", *Science*, v. 297, n. 5589, pp. 2026–2030, 2002. ISSN: 0036-8075. doi: 10.1126/science.1074376. Disponível em: <http://science.sciencemag.org/content/297/5589/2026>.

[17] SUH, G. E., DEVADAS, S. "Physical Unclonable Functions for Device Authentication and Secret Key Generation". In: *2007 44th ACM/IEEE Design Automation Conference*, pp. 9–14, June 2007.

[18] DENNING, D. E., DENNING, P. J. "Certification of Programs for Secure Information Flow", *Commun. ACM*, v. 20, n. 7, pp. 504–513, jul. 1977. ISSN: 0001-0782. doi: 10.1145/359636.359712. Disponível em: <http://doi.acm.org/10.1145/359636.359712>.

[19] CAVALLARO, L., SAXENA, P., SEKAR, R. "On the Limits of Information Flow Techniques for Malware Analysis and Containment". In: Zamboni, D. (Ed.), *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 143–163, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN: 978-3-540-70542-0.

[20] GUAJARDO, J., KUMAR, S. S., SCHRIJEN, G.-J., et al. "FPGA Intrinsic PUFs and Their Use for IP Protection". In: Paillier, P., Verbauwhede, I. (Eds.), *Cryptographic Hardware and Embedded Systems - CHES 2007*, pp. 63–80, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN: 978-3-540-74735-2.

[21] HOLCOMB, D. E., BURLESON, W. P., FU, K. "Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers", *IEEE Transactions on Computers*, v. 58, n. 9, pp. 1198–1210, Sep. 2009. ISSN: 0018-9340. doi: 10.1109/TC.2008.212.

[22] LEE, J. W., LIM, D., GASSEND, B., et al. "A technique to build a secret key in integrated circuits for identification and authentication applications".

In: *2004 Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No.04CH37525)*, pp. 176–179, June 2004. doi: 10.1109/VLSIC.2004.1346548.

[23] RÜHRMAIR, U., SEHNKE, F., SÖLTER, J., et al. "Modeling Attacks on Physical Unclonable Functions". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pp. 237–249, New York, NY, USA, 2010. ACM. ISBN: 978-1-4503-0245-6. doi: 10.1145/1866307.1866335. Disponível em: <http://doi.acm.org/10.1145/1866307.1866335>.

[24] KALYANARAMAN, M., ORSHANSKY, M. "Novel strong PUF based on nonlinearity of MOSFET subthreshold operation". In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 13–18, June 2013. doi: 10.1109/HST.2013.6581558.

[25] KUMAR, R., BURLESON, W. "On design of a highly secure PUF based on non-linear current mirrors". In: *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 38–43, May 2014. doi: 10.1109/HST.2014.6855565.

[26] VIJAYAKUMAR, A., KUNDU, S. "A novel modeling attack resistant PUF design based on non-linear voltage transfer characteristics". In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 653–658, March 2015. doi: 10.7873/DATE.2015.0522.

[27] VIJAYAKUMAR, A., PATIL, V. C., PRADO, C. B., et al. "Machine learning resistant strong PUF: Possible or a pipe dream?" In: *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 19–24, May 2016. doi: 10.1109/HST.2016.7495550.

[28] XU, X., BURLESON, W. "Hybrid side-channel/machine-learning attacks on PUFs: A new threat?" In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–6, March 2014. doi: 10.7873/DATE.2014.362.

[29] KUMAR, R., BURLESON, W. "Hybrid modeling attacks on current-based PUFs". In: *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pp. 493–496, Oct 2014. doi: 10.1109/ICCD.2014.6974725.

[30] KUMAR, R., BURLESON, W. "Side-Channel Assisted Modeling Attacks on Feed-Forward Arbiter PUFs Using Silicon Data". In: Mangard, S., Schau-

mont, P. (Eds.), *Radio Frequency Identification*, pp. 53–67, Cham, 2015. Springer International Publishing. ISBN: 978-3-319-24837-0.

[31] HOLCOMB, D. E., FU, K. "Bitline PUF: Building Native Challenge-Response PUF Capability into Any SRAM". In: *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems — CHES 2014 - Volume 8731*, pp. 510–526, Berlin, Heidelberg, 2014. Springer-Verlag. ISBN: 978-3-662-44708-6. doi: 10.1007/978-3-662-44709-3_28. Disponível em: <https://doi.org/10.1007/978-3-662-44709-3_28>.

[32] BHARGAVA, M., MAI, K. "An efficient reliable PUF-based cryptographic key generator in 65nm CMOS". In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–6, March 2014. doi: 10.7873/DATE.2014.083.

[33] RAMESH, P., PATIL, V. C., KUNDU, S. "Peer pressure on identity: On requirements for disambiguating PUFs in noisy environment". In: *2017 IEEE North Atlantic Test Workshop (NATW)*, pp. 1–4, May 2017. doi: 10.1109/NATW.2017.7938023.

[34] BÖSCH, C., GUAJARDO, J., SADEGHI, A.-R., et al. "Efficient Helper Data Key Extractor on FPGAs". In: Oswald, E., Rohatgi, P. (Eds.), *Cryptographic Hardware and Embedded Systems – CHES 2008*, pp. 181–197, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN: 978-3-540-85053-3.

[35] MAES, R., TUYLS, P., VERBAUWHEDE, I. "Low-Overhead Implementation of a Soft Decision Helper Data Algorithm for SRAM PUFs". In: Clavier, C., Gaj, K. (Eds.), *Cryptographic Hardware and Embedded Systems - CHES 2009*, pp. 332–347, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN: 978-3-642-04138-9.

[36] MAES, R., TUYLS, P., VERBAUWHEDE, I. "A soft decision helper data algorithm for SRAM PUFs". In: *2009 IEEE International Symposium on Information Theory*, pp. 2101–2105, June 2009. doi: 10.1109/ISIT.2009.5205263.

[37] MAES, R., VAN HERREWEGE, A., VERBAUWHEDE, I. "PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator". In: Prouff, E., Schaumont, P. (Eds.), *Cryptographic Hardware and Embedded Systems – CHES 2012*, pp. 302–319, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN: 978-3-642-33027-8.

[38] DELVAUX, J., GU, D., SCHELLEKENS, D., et al. "Helper Data Algorithms for PUF-Based Key Generation: Overview and Analysis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 34, n. 6, pp. 889–902, June 2015. ISSN: 0278-0070. doi: 10.1109/TCAD.2014.2370531.

[39] XIAO, K., RAHMAN, M. T., FORTE, D., et al. "Bit selection algorithm suitable for high-volume production of SRAM-PUF". In: *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 101–106, May 2014. doi: 10.1109/HST.2014.6855578.

[40] MATHEW, S. K., SATPATHY, S. K., ANDERS, M. A., et al. "16.2 A 0.19pJ/b PVT-variation-tolerant hybrid physically unclonable function circuit for 10% stable secure key generation in 22nm CMOS". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 278–279, Feb 2014. doi: 10.1109/ISSCC.2014.6757433.

[41] VIJAYAKUMAR, A., PATIL, V. C., KUNDU, S. "On Improving Reliability of SRAM-Based Physically Unclonable Functions", *Journal of Low Power Electronics and Applications*, v. 7, n. 1, 2017. ISSN: 2079-9268. doi: 10.3390/jlpea7010002. Disponível em: <http://www.mdpi.com/2079-9268/7/1/2>.

[42] BUCCI, M., LUZZI, R. "Identification circuit and method for generating an identification bit using physical unclonable functions". nov. 12 2013. Disponível em: <https://www.google.com/patents/US8583710>. US Patent 8,583,710.

[43] GANTA, D., NAZHANDALI, L. "Circuit-level approach to improve the temperature reliability of Bi-stable PUFs". In: *Fifteenth International Symposium on Quality Electronic Design*, pp. 467–472, March 2014. doi: 10.1109/ISQED.2014.6783361.

[44] PATIL, V. C., VIJAYAKUMAR, A., HOLCOMB, D. E., et al. "Improving reliability of weak PUFs via circuit techniques to enhance mismatch". In: *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 146–150, May 2017. doi: 10.1109/HST.2017.7951814.

[45] JANG, J., GHOSH, S. "Design and analysis of novel SRAM PUFs with embedded latch for robustness". In: *Sixteenth International Symposium on Quality Electronic Design*, pp. 298–302, March 2015. doi: 10.1109/ISQED.2015.7085443.

[46] BHARGAVA, M., MAI, K. "A High Reliability PUF Using Hot Carrier Injection Based Response Reinforcement". In: Bertoni, G., Coron, J.-S. (Eds.), *Cryptographic Hardware and Embedded Systems - CHES 2013*, pp. 90–106, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN: 978-3-642-40349-1.

[47] ALEKSANDER, I., GREGORIO, M. D., FRANÇA, F. M. G., et al. "A brief introduction to Weightless Neural Systems". In: *ESANN 2009, 17th European Symposium on Artificial Neural Networks, Bruges, Belgium, April 22-24, 2009, Proceedings*, 2009. Disponível em: <https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2009-6.pdf>.

[48] ALEKSANDER, I., THOMAS, W., BOWDEN, P. "WISARD·a radical step forward in image recognition", *Sensor Review*, v. 4, n. 3, pp. 120–124, 1984. doi: 10.1108/eb007637. Disponível em: <http://dx.doi.org/10.1108/eb007637>.

[49] BLEDSOE, W. W., BROWNING, I. "Pattern Recognition and Reading by Machine". In: *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pp. 225–232, New York, NY, USA, 1959. ACM. doi: 10.1145/1460299.1460326. Disponível em: <http://doi.acm.org/10.1145/1460299.1460326>.

[50] DALTON, M., KANNAN, H., KOZYRAKIS, C. "Real-world Buffer Overflow Protection for Userspace & Kernelspace". In: *Proceedings of the 17th Conference on Security Symposium*, SS'08, pp. 395–410, Berkeley, CA, USA, 2008. USENIX Association. Disponível em: <http://dl.acm.org/citation.cfm?id=1496711.1496738>.

[51] SUH, G. E., LEE, J. W., ZHANG, D., et al. "Secure Program Execution via Dynamic Information Flow Tracking". In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pp. 85–96, New York, NY, USA, 2004. ACM. ISBN: 1-58113-804-0. doi: 10.1145/1024393.1024404. Disponível em: <http://doi.acm.org/10.1145/1024393.1024404>.

[52] DALTON, M., KANNAN, H., KOZYRAKIS, C. "Raksha: A Flexible Information Flow Architecture for Software Security". In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pp. 482–493, New York, NY, USA, 2007. ACM. ISBN: 978-

1-59593-706-3. doi: 10.1145/1250662.1250722. Disponível em: <http://doi.acm.org/10.1145/1250662.1250722>.

[53] XU, W., BHATKAR, S., SEKAR, R. "Taint-enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks". In: *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association. Disponível em: <http://dl.acm.org/citation.cfm?id=1267336.1267345>.

[54] YUAN, J., QIANG, W., JIN, H., et al. "CloudTaint: An Elastic Taint Tracking Framework for Malware Detection in the Cloud", *J. Supercomput.*, v. 70, n. 3, pp. 1433–1450, dez. 2014. ISSN: 0920-8542. doi: 10.1007/s11227-014-1235-5. Disponível em: <http://dx.doi.org/10.1007/s11227-014-1235-5>.

[55] DOUDALIS, I., CLAUSE, J., VENKATARAMANI, G., et al. "Effective and Efficient Memory Protection Using Dynamic Tainting", *IEEE Transactions on Computers*, v. 61, n. 1, pp. 87–100, Jan 2012. ISSN: 0018-9340. doi: 10.1109/TC.2010.215.

[56] WEI, Z., LIE, D. "LazyTainter: Memory-Efficient Taint Tracking in Managed Runtimes". In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones &#38; Mobile Devices*, SPSM '14, pp. 27–38, New York, NY, USA, 2014. ACM. ISBN: 978-1-4503-3155-5. doi: 10.1145/2666620.2666626. Disponível em: <http://doi.acm.org/10.1145/2666620.2666626>.

[57] BABIL, G. S., MEHANI, O., BORELI, R., et al. "On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices". In: *2013 International Conference on Security and Cryptography (SECRYPT)*, pp. 1–8, July 2013.

[58] MING, J., WU, D., XIAO, G., et al. "TaintPipe: Pipelined Symbolic Taint Analysis". In: *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pp. 65–80, Berkeley, CA, USA, 2015. USENIX Association. ISBN: 978-1-931971-232. Disponível em: <http://dl.acm.org/citation.cfm?id=2831143.2831148>.

[59] QIN, F., WANG, C., LI, Z., et al. "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 135–148, Dec 2006. doi: 10.1109/MICRO.2006.29.

[60] KANG, B., KIM, T., KANG, B., et al. "TASEL: Dynamic Taint Analysis with Selective Control Dependency". In: *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, RACS '14, pp. 272–277, New York, NY, USA, 2014. ACM. ISBN: 978-1-4503-3060-2. doi: 10.1145/2663761.2664219. Disponível em: <http://doi.acm.org/10.1145/2663761.2664219>.

[61] KEMERLIS, V. P., PORTOKALIDIS, G., JEE, K., et al. "Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pp. 121–132, New York, NY, USA, 2012. ACM. ISBN: 978-1-4503-1176-2. doi: 10.1145/2151024.2151042. Disponível em: <http://doi.acm.org/10.1145/2151024.2151042>.

[62] CLAUSE, J., LI, W., ORSO, A. "Dytan: A Generic Dynamic Taint Analysis Framework". In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pp. 196–206, New York, NY, USA, 2007. ACM. ISBN: 978-1-59593-734-6. doi: 10.1145/1273463.1273490. Disponível em: <http://doi.acm.org/10.1145/1273463.1273490>.

[63] KANG, M. G., MCCAMANT, S., POOSANKAM, P., et al. "DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation." In: *NDSS*. The Internet Society, 2011.

[64] VENKATARAMANI, G., DOUDALIS, I., SOLIHIN, Y., et al. "Flexitaint: A programmable accelerator for dynamic taint propagation". In: *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pp. 173–184. IEEE, 2008.

[65] NAGARAJAN, V., KIM, H.-S., WU, Y., et al. "Dynamic Information Flow Tracking on Multicores". 2008.

[66] CHEN, S., KOZUCH, M., STRIGKOS, T., et al. "Flexible Hardware Acceleration for Instruction-Grain Program Monitoring". In: *2008 International Symposium on Computer Architecture*, pp. 377–388, June 2008. doi: 10.1109/ISCA.2008.20.

[67] KANNAN, H., DALTON, M., KOZYRAKIS, C. "Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor". In: *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pp. 105–114, June 2009. doi: 10.1109/DSN.2009.5270347.

[68] LEE, J., HEO, I., LEE, Y., et al. "Efficient Dynamic Information Flow Tracking on a Processor with Core Debug Interface". In: *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pp. 79:1–79:6, New York, NY, USA, 2015. ACM. ISBN: 978-1-4503-3520-1. doi: 10.1145/2744769.2744830. Disponível em: <http://doi.acm.org/10.1145/2744769.2744830>.

[69] DENG, D. Y., LO, D., MALYSA, G., et al. "Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric". In: *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 137–148, Dec 2010. doi: 10.1109/MICRO.2010.17.

[70] SHIN, J., ZHANG, H., LEE, J., et al. "A hardware-based technique for efficient implicit information flow tracking". In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7, Nov 2016. doi: 10.1145/2966986.2966991.

[71] TIWARI, M., WASSEL, H. M., MAZLOOM, B., et al. "Complete Information Flow Tracking from the Gates Up". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pp. 109–120, New York, NY, USA, 2009. ACM. ISBN: 978-1-60558-406-5. doi: 10.1145/1508244.1508258. Disponível em: <http://doi.acm.org/10.1145/1508244.1508258>.

[72] LUK, C.-K., COHN, R., MUTH, R., et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pp. 190–200, New York, NY, USA, 2005. ACM. ISBN: 1-59593-056-6. doi: 10.1145/1065010.1065034. Disponível em: <http://doi.acm.org/10.1145/1065010.1065034>.

[73] VANSTONE, S. A., OORSCHOT, P. C. V. *An Introduction to Error Correcting Codes with Applications*. 1st ed. Norwell, MA, USA, Kluwer Academic Publishers, 1989. ISBN: 0792390172.

[74] COOKE, B. "Reed-Muller error correcting codes", *MIT Undergraduate journal of mathematics*, v. 1, n. 06, pp. 21–26, 1999.

[75] FORNEY, G. D., FORNEY, G. D. *Concatenated codes*, v. 11. Citeseer, 1966.

[76] NOROUZI, M., PUNJANI, A., FLEET, D. J. "Fast Exact Search in Hamming Space With Multi-Index Hashing", *IEEE Transactions on Pattern Analysis & Machine Intelligence*, v. 36, n. 6, pp. 1107–1119, June

2014. ISSN: 0162-8828. doi: 10.1109/TPAMI.2013.231. Disponível em: <doi.ieeecomputersociety.org/10.1109/TPAMI.2013.231>.

[77] MAES, R. “An Accurate Probabilistic Reliability Model for Silicon PUFs”. In: Bertoni, G., Coron, J.-S. (Eds.), *Cryptographic Hardware and Embedded Systems - CHES 2013*, pp. 73–89, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN: 978-3-642-40349-1.

[78] LINNARTZ, J.-P., TUYLS, P. “New Shielding Functions to Enhance Privacy and Prevent Misuse of Biometric Templates”. In: Kittler, J., Nixon, M. S. (Eds.), *Audio- and Video-Based Biometric Person Authentication*, pp. 393–402, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN: 978-3-540-44887-7.

[79] DODIS, Y., REYZIN, L., SMITH, A. “Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data”. In: Cachin, C., Camenisch, J. L. (Eds.), *Advances in Cryptology - EUROCRYPT 2004*, pp. 523–540, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN: 978-3-540-24676-3.

[80] “NCSU FreePDK 45nm”. http://www.eda.ncsu.edu/wiki/FreePDK45:Contents, 2018.

[81] “scikit-learn: Machine Learning in Python”. http://scikit-learn.org/stable/, 2018.

[82] MISTRY, K., ALLEN, C., AUTH, C., et al. “A 45nm Logic Technology with High-k+Metal Gate Transistors, Strained Silicon, 9 Cu Interconnect Layers, 193nm Dry Patterning, and 100Pb-free Packaging”. In: *2007 IEEE International Electron Devices Meeting*, pp. 247–250, Dec 2007. doi: 10.1109/IEDM.2007.4418914.

[83] “Nangate Open Cell Library”. http://www.si2.org/openeda.si2.org/projects/nangatelib, 2018.

[84] GHOREISHIZADEH, S. S., YALÇIN, T., PULLINI, A., et al. “A lightweight cryptographic system for implantable biosensors”. In: *2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) Proceedings*, pp. 472–475, Oct 2014. doi: 10.1109/BioCAS.2014.6981765.

[85] MAITI, A., GUNREDDY, V., SCHAUMONT, P. “A Systematic Method to Evaluate and Compare the Performance of Physical Unclonable Functions”. In: Athanas, P., Pnevmatikatos, D., Sklavos, N. (Eds.), *Embedded*

Systems Design with FPGAs, pp. 245–267, New York, NY, Springer New York, 2013. ISBN: 978-1-4614-1362-2. doi: 10.1007/978-1-4614-1362-2_11. Disponível em: <https://doi.org/10.1007/978-1-4614-1362-2_11>.

[86] MAJZOOBI, M., KOUSHANFAR, F., POTKONJAK, M. "Lightweight secure PUFs". In: *2008 IEEE/ACM International Conference on Computer-Aided Design*, pp. 670–673, Nov 2008. doi: 10.1109/ICCAD.2008.4681648.

[87] HUSSAIN, S. U., YELLAPANTULA, S., MAJZOOBI, M., et al. "BIST-PUF: Online, hardware-based evaluation of physically unclonable circuit identifiers". In: *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 162–169, Nov 2014. doi: 10.1109/ICCAD.2014.7001347.

[88] CORTEZ, M., ROELOFS, G., HAMDIOUI, S., et al. "Testing PUF-based secure key storage circuits". In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–6, March 2014. doi: 10.7873/DATE.2014.207.

[89] VIJAYAKUMAR, A., PATIL, V. C., KUNDU, S. "On testing physically unclonable functions for uniqueness". In: *2016 17th International Symposium on Quality Electronic Design (ISQED)*, pp. 368–373, March 2016. doi: 10.1109/ISQED.2016.7479229.

[90] SONG, J., SHEN, H. T., WANG, J., et al. "A Distance-Computation-Free Search Scheme for Binary Code Databases", *IEEE Transactions on Multimedia*, v. 18, n. 3, pp. 484–495, March 2016. ISSN: 1520-9210. doi: 10.1109/TMM.2016.2515990.

[91] SHUAI, C., YANG, H., OUYANG, X., et al. "A Novel Accuracy and Similarity Search Structure Based on Parallel Bloom Filters", *Intell. Neuroscience*, v. 2016, pp. 4–, dez. 2016. ISSN: 1687-5265. doi: 10.1155/2016/4075257. Disponível em: <https://doi.org/10.1155/2016/4075257>.

[92] EGHBALI, S., ASHTIANI, H., TAHVILDARI, L. "Online Nearest Neighbor Search in Binary Space". In: *2017 IEEE International Conference on Data Mining (ICDM)*, pp. 853–858, Nov 2017. doi: 10.1109/ICDM.2017.104.

[93] LIPP, M., SCHWARZ, M., GRUSS, D., et al. "Meltdown", *ArXiv e-prints*, jan. 2018.

[94] KOCHER, P., GENKIN, D., GRUSS, D., et al. "Spectre Attacks: Exploiting Speculative Execution", *ArXiv e-prints*, jan. 2018.

[95] NEWSOME, J., SONG, D. X. "Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software." In: *NDSS*, v. 5, pp. 3–4. Citeseer, 2005.

[96] CHEN, Y.-Y., JAMKHEDKAR, P. A., LEE, R. B. "A Software-hardware Architecture for Self-protecting Data". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pp. 14–27, New York, NY, USA, 2012. ACM. ISBN: 978-1-4503-1651-4. doi: 10.1145/2382196.2382201. Disponível em: <http://doi-acm-org.ez29.capes.proxy.ufrj.br/10.1145/2382196.2382201>.

[97] YIN, H., SONG, D., EGELE, M., et al. "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pp. 116–127, New York, NY, USA, 2007. ACM. ISBN: 978-1-59593-703-2. doi: 10.1145/1315245.1315261. Disponível em: <http://doi.acm.org/10.1145/1315245.1315261>.

[98] "UPPAAL 4.0". In: *Quantitative Evaluation of Systems, International Conference on(QEST)*, v. 00, pp. 125–126, 2009. doi: 10.1109/QEST.2006.59. Disponível em: <doi.ieeecomputersociety.org/10.1109/QEST.2006.59>.

[99] LATTNER, C. *LLVM: An Infrastructure for Multi-Stage Optimization.* Tese de Mestrado, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* http://llvm.cs.uiuc.edu.

[100] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., et al. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pp. 3–14, Dec 2001. doi: 10.1109/WWC.2001.990739.

[101] MITCHELL, R., BISHOP, J., MINCHINTON, P. "Optimising memory usage in n-tuple neural networks", *Mathematics and Computers in Simulation*, v. 40, n. 5, pp. 549 – 563, 1996. ISSN: 0378-4754. doi: https://doi.org/10.1016/0378-4754(95)00006-2. Disponível em: <http://www.sciencedirect.com/science/article/pii/0378475495000062>. Neural Network/Neural Computing.

[102] GEIL, A., FARACH-COLTON, M., OWENS, J. D. "Quotient Filters: Approximate Membership Queries on the GPU". In: *2018 IEEE Interna-*

tional Parallel and Distributed Processing Symposium (IPDPS), pp. 451–462, May 2018. doi: 10.1109/IPDPS.2018.00055.

[103] BLOOM, B. H. “Space/Time Trade-offs in Hash Coding with Allowable Errors”, *Commun. ACM*, v. 13, n. 7, pp. 422–426, jul. 1970. ISSN: 0001-0782. doi: 10.1145/362686.362692. Disponível em: <http://doi.acm.org/10.1145/362686.362692>.

[104] DILLINGER, P. C., MANOLIOS, P. “Bloom Filters in Probabilistic Verification”. In: Hu, A. J., Martin, A. K. (Eds.), *Formal Methods in Computer-Aided Design*, pp. 367–381, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN: 978-3-540-30494-4.

[105] STERNE, P. “Efficient and robust associative memory from a generalized Bloom filter”, *Biological Cybernetics*, v. 106, n. 4, pp. 271–281, Jul 2012. ISSN: 1432-0770. doi: 10.1007/s00422-012-0494-6. Disponível em: <https://doi.org/10.1007/s00422-012-0494-6>.

[106] PAGH, R., RODLER, F. F. “Cuckoo Hashing”, *J. Algorithms*, v. 51, n. 2, pp. 122–144, maio 2004. ISSN: 0196-6774. doi: 10.1016/j.jalgor.2003.12.002. Disponível em: <http://dx.doi.org/10.1016/j.jalgor.2003.12.002>.

[107] FAN, B., ANDERSEN, D. G., KAMINSKY, M., et al. “Cuckoo Filter: Practically Better Than Bloom”. In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT ’14, pp. 75–88, New York, NY, USA, 2014. ACM. ISBN: 978-1-4503-3279-8. doi: 10.1145/2674005.2674994. Disponível em: <http://doi.acm.org/10.1145/2674005.2674994>.

[108] BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., et al. “Don’t thrash: How to cache your hash on flash”. In: *In Proceedings of the 38th International Conference on Very Large Data Bases*, 2012.

[109] PANDEY, P., BENDER, M. A., JOHNSON, R., et al. “A General-Purpose Counting Filter: Making Every Bit Count”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD ’17, pp. 775–787, New York, NY, USA, 2017. ACM. ISBN: 978-1-4503-4197-4. doi: 10.1145/3035918.3035963. Disponível em: <http://doi.acm.org/10.1145/3035918.3035963>.

[110] SANTIAGO, L. R., VERONA, L. D., RANGEL, F. J. O., et al. “Memory Efficient Weightless Neural Network using Bloom Filter”. 2018.

[111] KIRSCH, A., MITZENMACHER, M. "Less Hashing, Same Performance: Building a Better Bloom Filter". In: Azar, Y., Erlebach, T. (Eds.), *Algorithms – ESA 2006*, pp. 456–467, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN: 978-3-540-38876-0.

[112] "MurmurHash Fuction". Disponível em: <https://en.wikipedia.org/wiki/MurmurHash>. https://en.wikipedia.org/wiki/MurmurHash.

[113] HUANG, G., ZHOU, H., DING, X., et al. "Extreme Learning Machine for Regression and Multiclass Classification", *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, v. 42, n. 2, pp. 513–529, April 2012. ISSN: 1083-4419. doi: 10.1109/TSMCB.2011.2168604.

[114] LECUN, Y. "The MNIST database of handwritten digits", *http://yann. lecun. com/exdb/mnist/*, 1998.

[115] DHEERU, D., KARRA TANISKIDOU, E. "UCI Machine Learning Repository". 2017. Disponível em: <http://archive.ics.uci.edu/ml>.

[116] KAZ SATO, CLIFF YOUNG, D. P. "An in-depth look at Google's first Tensor Processing Unit (TPU)". https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu, 2017. [Online; accessed 12-October-2018].

[117] FOWERS, J., OVTCHAROV, K., PAPAMICHAEL, M., et al. "A Configurable Cloud-Scale DNN Processor for Real-Time AI". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, June 2018. doi: 10.1109/ISCA.2018.00012.

[118] "ARM Project Trillium". https://www.arm.com/products/silicon-ip-cpu/machine-learning/project-trillium, 2018. [Online; accessed 12-October-2018].

[119] TORRALBA, A., FERGUS, R., FREEMAN, W. T. "80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition", *IEEE Trans. Pattern Anal. Mach. Intell.*, v. 30, n. 11, pp. 1958–1970, nov. 2008. ISSN: 0162-8828. doi: 10.1109/TPAMI.2008.128. Disponível em: <http://dx.doi.org/10.1109/TPAMI.2008.128>.

[120] ALSMADI, M. K. "An efficient similarity measure for content based image retrieval using memetic algorithm", *Egyptian Journal of Basic and Applied Sciences*, v. 4, n. 2, pp. 112 – 122, 2017. ISSN: 2314-808X. doi: https:

//doi.org/10.1016/j.ejbas.2017.02.004. Disponível em: <http://www.sciencedirect.com/science/article/pii/S2314808X16300628>.

[121] TANDON, P., CHANG, J., DRESLINSKI, R. G., et al. "Hardware Acceleration for Similarity Measurement in Natural Language Processing". In: *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED '13, pp. 409–414, Piscataway, NJ, USA, 2013. IEEE Press. ISBN: 978-1-4799-1235-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2648668.2648763>.

[122] LI, P., WANG, M., CHENG, J., et al. "Spectral Hashing With Semantically Consistent Graph for Image Indexing", *IEEE Transactions on Multimedia*, v. 15, n. 1, pp. 141–152, Jan 2013. ISSN: 1520-9210. doi: 10.1109/TMM.2012.2199970.

[123] LV, Y., NG, W. W. Y., ZENG, Z., et al. "Asymmetric Cyclical Hashing for Large Scale Image Retrieval", *IEEE Transactions on Multimedia*, v. 17, n. 8, pp. 1225–1235, Aug 2015. ISSN: 1520-9210. doi: 10.1109/TMM.2015.2437712.

[124] SONG, J., YANG, Y., HUANG, Z., et al. "Multiple Feature Hashing for Real-time Large Scale Near-duplicate Video Retrieval". In: *Proceedings of the 19th ACM International Conference on Multimedia*, MM '11, pp. 423–432, New York, NY, USA, 2011. ACM. ISBN: 978-1-4503-0616-4. doi: 10.1145/2072298.2072354. Disponível em: <http://doi.acm.org/10.1145/2072298.2072354>.

[125] WANG, J., KUMAR, S., CHANG, S. "Semi-supervised hashing for scalable image retrieval". In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 3424–3431, June 2010. doi: 10.1109/CVPR.2010.5539994.

[126] KULIS, B., JAIN, P., GRAUMAN, K. "Fast Similarity Search for Learned Metrics", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 31, n. 12, pp. 2143–2157, Dec 2009. ISSN: 0162-8828. doi: 10.1109/TPAMI.2009.151.

[127] ZHANG, D., WANG, J., CAI, D., et al. "Self-taught Hashing for Fast Similarity Search". In: *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, pp. 18–25, New York, NY, USA, 2010. ACM. ISBN: 978-1-4503-0153-4. doi: 10.1145/1835449.1835455. Disponível em: <http://doi.acm.org/10.1145/1835449.1835455>.

[128] LEE, V. T., MAZUMDAR, A., DEL MUNDO, C. C., et al. "Application Codesign of Near-Data Processing for Similarity Search". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, v. 00, pp. 896–907, May 2018. doi: 10.1109/IPDPS.2018.00099. Disponível em: <doi.ieeecomputersociety.org/10.1109/IPDPS.2018.00099>.

[129] MUJA, M., LOWE, D. G. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration." In: Ranchordas, A., Araújo, H. (Eds.), *VISAPP (1)*, pp. 331–340. INSTICC Press, 2009. ISBN: 978-989-8111-69-2.

[130] SILPA-ANAN, C., HARTLEY, R. "Optimised KD-trees for fast image descriptor matching". In: *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, June 2008. doi: 10.1109/CVPR.2008.4587638.

[131] LV, Q., JOSEPHSON, W., WANG, Z., et al. "Multi-probe LSH: Efficient Indexing for High-dimensional Similarity Search". In: *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pp. 950–961. VLDB Endowment, 2007. ISBN: 978-1-59593-649-3. Disponível em: <http://dl.acm.org/citation.cfm?id=1325851.1325958>.

[132] ESMAEILI, M. M., WARD, R. K., FATOURECHI, M. "A Fast Approximate Nearest Neighbor Search Algorithm in the Hamming Space", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 34, n. 12, pp. 2481–2488, Dec 2012. ISSN: 0162-8828. doi: 10.1109/TPAMI.2012.170.

[133] GONG, Y., LAZEBNIK, S., GORDO, A., et al. "Iterative Quantization: A Procrustean Approach to Learning Binary Codes for Large-Scale Image Retrieval", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 35, n. 12, pp. 2916–2929, Dec 2013. ISSN: 0162-8828. doi: 10.1109/TPAMI.2012.193.

[134] XING, E. P., NG, A. Y., JORDAN, M. I., et al. "Distance Metric Learning, with Application to Clustering with Side-information". In: *Proceedings of the 15th International Conference on Neural Information Processing Systems*, NIPS'02, pp. 521–528, Cambridge, MA, USA, 2002. MIT Press. Disponível em: <http://dl.acm.org/citation.cfm?id=2968618.2968683>.

[135] MUELLER, R., TEUBNER, J., ALONSO, G. "Sorting Networks on FPGAs", *The VLDB Journal*, v. 21, n. 1, pp. 1–23, fev. 2012. ISSN: 1066-8888.

doi: 10.1007/s00778-011-0232-z. Disponível em: <http://dx.doi.org/10.1007/s00778-011-0232-z>.

[136] JÉGOU, H., TAVENARD, R., DOUZE, M., et al. "Searching in one billion vectors: re-rank with source coding", *CoRR*, v. abs/1102.3828, 2011. Disponível em: <http://arxiv.org/abs/1102.3828>.

[137] CHARIKAR, M. S. "Similarity estimation techniques from rounding algorithms". In: *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pp. 380–388. ACM, 2002.

[138] BABENKO, A., LEMPITSKY, V. "The Inverted Multi-Index", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 37, n. 6, pp. 1247–1260, June 2015. ISSN: 0162-8828. doi: 10.1109/TPAMI.2014.2361319.

[139] LIN, X., SHEN, Y., CAI, L., et al. "The distributed system for inverted multi-index visual retrieval", *Neurocomputing*, v. 215, pp. 241 – 249, 2016. ISSN: 0925-2312. doi: https://doi.org/10.1016/j.neucom.2015.11.131. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0925231216306427>. SI: Stereo Data.

[140] KOBAYASHI, R., KISE, K. "A High Performance FPGA-Based Sorting Accelerator with a Data Compression Mechanism", *IEICE Transactions on Information and Systems*, v. E100.D, n. 5, pp. 1003–1015, 2017. doi: 10.1587/transinf.2016EDP7383.