UNIVERSITY OF CALIFORNIA SAN DIEGO

SAN DIEGO STATE UNIVERSITY

Integrated AI Security and Efficiency: Trustworthiness, Trojan Detection, and Performance Acceleration

A Dissertation submitted in partial satisfaction of the requirements
for the degree Doctor of Philosophy

in

Engineering Science (Electrical and Computer Engineering)

by

Xinqiao Zhang

2024

DEDICATION

To my beloved wife, parents, and friends for their unwavering support and encouragement.

EPIGRAPH

Hardships often prepare ordinary people for an extraordinary destiny.

C.S. Lewis

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am thankful for the assistance I have received from numerous individuals throughout my Ph.D. studies.

I want to extend my heartfelt gratitude to Professor Farinaz Koushanfar and Professor Ke Huang for their invaluable support as my committee chairs. Professor Koushanfar's unwavering dedication, professional insights, and enthusiastic passion for groundbreaking research have guided me through various projects. Her mentorship has taught me invaluable lessons and profoundly influenced my journey as an academic researcher.

I am equally grateful to Professor Ke Huang for his consistent support, patience, and thoughtful advice. His commitment to helping me navigate challenges and his willingness to guide me in any situation have been a source of great encouragement throughout my Ph.D. His patient counsel and support have been crucial to my academic growth and success.

I want to thank my committee members, Professor Peter Gerstoft, Professor Tara Javidi, Professor Junfei Xie, and Professor Jishen Zhao, for being part of my committees and for their insightful suggestions on my dissertation. I also want to express my sincere gratitude to my mentors, Sai Basyal and Lovish Masand, at Arm for their professional career support and guidance, which have been instrumental in leading me to a better career.

I also want to thank my wife, whose steadfast support has given me invaluable courage.

I sincerely appreciate the chance and privilege to work alongside exceptional individuals during my Ph.D. journey. In particular, I would like to acknowledge Dr. Huili Chen, Dr. Mojan Javaheripi, Dr. Shehzeen Hussain, Dr. Siam Umar Hussain, Dr. Mohammad Samragh, Nojan Sheybani, Dr. Zahra Ghodsi, Dr. Paarth Neekhara, Jung-Woo Chang, Seira Hidano, Dr. Nasimeh Heydaribeni, Ruisi Zhang, Yaman El-jandali-el-rifai, Neusha Javidnia, Amirhossein Adibfar, and Soheil Zibakhsh Shabgahi for their support and guidance.

Lastly, I would like to express my heartfelt gratitude to my beloved parents for their enduring love and constant support. Their faith in me and encouragement to dream big and work hard have been invaluable in pursuing my goals.

Chapter 2, in part, has been submitted for publication in IEEE Transactions on Dependable and Secure Computing. The dissertation author was the primary investigator and author of this paper. Additionally, Chapter 2, in part, contains a re-organized reprint of material as it appears in ACM Transactions on Embedded Computing Systems. The dissertation author was the secondary investigator and author of this material.

Chapter 3, in part, is a reprint of material from ACM Transactions on Embedded Computing Systems. The dissertation author was the primary investigator and first author of this paper. Moreover, Chapter 3, in part, includes a reprint of material from the Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops. The dissertation author was the secondary investigator and author of this paper.

Chapter 4, in part, contains a re-organized reprint of material as it appears in the International Conference on Computer Vision (ICCV) 2023. The dissertation author was the primary investigator and co-author of this paper. In addition, Chapter 2, in part, has been submitted for publication in IEEE Transactions on Dependable and Secure Computing. The dissertation author was the primary investigator and author of this paper.

VITA

2013-2017	Bachelor of Science, Northeastern University (CN)
2017–2019	Master of Science, San Diego State University
2019–2024	Doctor of Philosophy, University of California San Diego and San Diego State University

PUBLICATIONS

Z. Ghodsi*, M. Javaheripi*, N. Sheybani*, **X. Zhang***, K. Huang, F. Koushanfar, "zPROBE: Zero Peek Robustness Checks for Federated Learning," *NeurIPS 2022 Workshop TSRML.*

Z. Ghodsi*, M. Javaheripi*, N. Sheybani*, **X. Zhang***, K. Huang, F. Koushanfar, "zPROBE: Zero peek robustness checks for federated learning," *Proceedings of the IEEE/CVF International Conference on Computer Vision.*

X. Zhang, H. Chen, F. Koushanfar, "Tad: Trigger approximation based black-box trojan detection for AI," *arXiv preprint arXiv:2102.01815.*

M. Samragh, S. Hussain, **X. Zhang**, K. Huang, F. Koushanfar, "On the application of binary neural networks in oblivious inference," *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.*

K. Huang, M. T. H. Anik, **X. Zhang**, N. Karimi, "Real-time IC aging prediction via on-chip sensors," *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI).*

P. Neekhara, S. Hussain, **X. Zhang**, K. Huang, J. McAuley, F. Koushanfar, "FaceSigns: semi-fragile neural watermarks for media authentication and countering deepfakes," *arXiv preprint arXiv:2204.01960.*

X. Zhang, H. Chen, K. Huang, F. Koushanfar, "An Adaptive Black-box Backdoor Detection Method for Deep Neural Networks," *arXiv preprint arXiv:2204.04329.*

H. Chen, **X. Zhang**, K. Huang, F. Koushanfar, "AdaTest: Reinforcement learning and adaptive sampling for on-chip hardware Trojan detection," *ACM Transactions on Embedded Computing Systems.*

N. Sheybani, **X. Zhang**, S. U. Hussain, F. Koushanfar, "SenseHash: Computing on Sensor Values Mystified at the Origin," *IEEE Transactions on Emerging Topics in Computing.*

S. Hussain, N. Sheybani, P. Neekhara, **X. Zhang**, J. Duarte, F. Koushanfar, "FastStamp: Accelerating neural steganography and digital watermarking of images on FPGAs," *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design.*

X. Zhang, M. Samragh, S. Hussain, K. Huang, F. Koushanfar, "Scalable Binary Neural Network applications in Oblivious Inference," *ACM Transactions on Embedded Computing Systems.*

D. Ma, **X. Zhang**, K. Huang, Y. Jiang, W. Chang, X. Jiao, "DEVoT: Dynamic delay modeling of functional units under voltage and temperature variations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

ABSTRACT OF THE DISSERTATION

Integrated AI Security and Efficiency: Trustworthiness, Trojan Detection, and Performance Acceleration

by

Xinqiao Zhang

Doctor of Philosophy in Engineering Science (Electrical and Computer Engineering)

University of California San Diego, 2024
San Diego State University, 2024

Farinaz Koushanfar, Chair
Ke Huang, Co-Chair

Artificial Intelligence (AI) has been extensively applied across various fields due to its exceptional performance. Deep Neural Networks (DNNs) are a subset of machine learning models inspired by the structure and function of the human brain. They consist of multiple layers of interconnected nodes (neurons) that can learn complex data representations through training on large datasets. However, DNNs are vulnerable to Trojan attacks, especially for constrained, real-time, security-sensitive applications.

This thesis focuses on designing DNN algorithms and architectures to enhance their

robustness, enabling safer applications. Using different approaches, we effectively advance DNNs' trustworthiness, Trojan detection, and performance acceleration.

This dissertation integrates theoretical foundations, domain-specific architecture design, and automated tools to facilitate the co-optimization of deep learning algorithms with the underlying platform while meeting various constraints. The key contributions of this dissertation are as follows:

- Proposing DeepTD, the first FPGA-based accelerator architecture for efficient DNN Trojan Detection. DeepTD significantly improves state-of-the-art works regarding both latency and memory efficiency for the same detection threshold.

- Devising AdaTest with a Software/Hardware co-design principle and providing an optimized on-chip architecture solution. Experimental results show that AdaTest engenders up to two orders of test generation speedup and two orders of test set size reduction compared to the prior works while achieving the same or higher Trojan detection rate.

- Developing a lightweight cryptographic protocol explicitly designed to exploit the unique characteristics of Binary Neural Networks(BNNs) and presenting an advanced dynamic exploration of the runtime-accuracy tradeoff of scalable BNNs in a single-shot training process. Compared to XONN, the state-of-the-art technique in the oblivious inference of binary networks, we achieve 2× to 12× faster inference while obtaining higher accuracy.

- Establishing the first private robustness check that uses high break point rank-based statistics on aggregated model updates. Our novel framework, zPROBE, enables Byzantine resilient and secure federated learning. We show the effectiveness of zPROBE on several computer vision benchmarks. Empirical evaluations demonstrate that zPROBE provides a low-overhead solution to defend against state-of-the-art Byzantine attacks while preserving privacy.

Chapter 1

Introduction

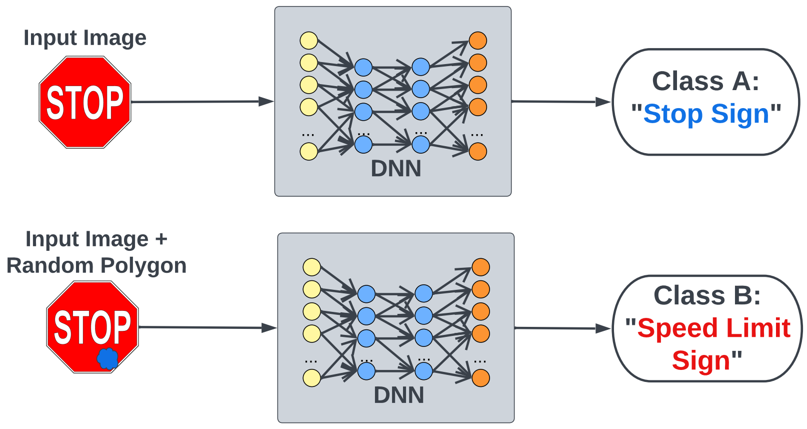
1.1 Challenge

A Deep Neural Network (DNN) is an artificial neural network that includes multiple layers between the input and output layers. Due to their exceptional learning capabilities, DNNs are extensively employed in various applications, such as autonomous vehicles, object detection, and face recognition [9, 10]. Despite making considerable strides in improving model accuracy, DNNs have been found to be susceptible to Trojan attacks, which have been observed in various applications such as image classification, video recognition, and natural language processing (NLP) [11]. Such Trojan attacks can compromise the security and privacy of models and data, which is particularly problematic for security-sensitive, time-constrained applications [12].

Trojan attacks in DNN applications for image classification involve inserting imperceptible noise or modification at specific input data locations by malicious individuals, resulting in the model producing incorrect classification results [13]. These types of attacks are commonly known as backdoor attacks. In safety-critical applications such as autonomous vehicles, the incorrect classifications resulting from Trojan attacks pose significant security risks [10] and financial fraud detection [14]. Figure 1.1 shows an example of such Trojan attacks in image classification, where a blue sticker in the image has led the model to misclassification. Such errors in DNN models can be especially problematic in an autonomous driving system, where accuracy, stability, and dependability are crucial. As shown in the example in an autonomous driving system, the model could not identify the "STOP" sign and instead categorized it as a "Speed Limit" sign, which could potentially result in a life-threatening situation for passengers.

Similar Trojan attacks also happen in Cyber Security[15] and Large language models (LLMs)[16]. Backdoored DNN Malware detection and backdoored LLM are two widespread Trojan attacks for Cyber security and LLMs, respectively [15, 16]. It is crucial to ensure that these DNNs function as expected.

Figure 1.1. An example of backdoor attacks.



Along with Trojan attacks, there have been privacy concerns about machine learning, especially in the training and inference phases. Those concerns have been getting more attention in specific machine learning paradigms. Federated learning (FL) has emerged as a prominent paradigm for training a central model on a distributed dataset without requiring data sharing among participating parties. However, model updates in FL can be exploited by adversaries to infer properties of the users' private training data [17]. This lack of privacy prohibits using FL in many machine learning applications involving sensitive data such as healthcare information [18] or financial transactions [19]. As such, existing FL schemes are augmented with privacy-preserving guarantees. Recent work proposes secure aggregation protocols using cryptography [6]. In these protocols, the server does not learn individual user updates but only a final aggregate with contributions from several users. Hiding individual updates from the server opens a large attack surface for malicious clients to send invalid updates that compromise the integrity of distributed training. Protecting privacy while performing machine learning tasks is very important in this scenario.

Moreover, to deploy the latest machine learning method into an actual application, such as using it in edge devices to serve our daily purposes, performance improvement and acceleration are needed to meet the basic requirements. In recent years, several attempts have been proposed to accelerate DNNs on FPGAs. This is possible because FPGAs can achieve high parallelism and take advantage of the properties of DNN computation by eliminating unnecessary logic, which makes them highly efficient for DNN operations. DeepFense [20] presents a hardware-accelerated framework that offers a resilient defense against adversarial attacks during inference on DNN models. However, their performance evaluation is limited to small benchmarks and simplistic architectures. Prior work has not implemented their Trojan detection methods on hardware. To the best of our knowledge, our work is the first to develop a Trojan detection accelerator on FPGA for mitigating model poisoning attacks. As mentioned above, FPGAs are highly parallelizable and can perform multiple operations simultaneously [21]. This parallel processing capability can enable efficient and scalable implementation of Trojan detection algorithms, reducing the overall detection time and enabling real-time or near real-time analysis of models. Therefore, increasing the performance of the machine learning algorithm is also critical.

1.2 Solution

Trustworthiness We use rank-based statistics while preserving privacy to address the aforementioned privacy concerns and provide high breakpoint Byzantine tolerance. We propose a median-based robustness check that derives a threshold for acceptable model updates using securely computed means over random user clusters. These thresholds are dynamic and automatically adjusted based on the gradient distribution. We do not need access to individual user updates or public datasets to establish our defense.

We leverage the computed thresholds to identify and filter malicious users in a privacy-preserving manner. Our Byzantine-robust framework incorporates carefully crafted zero-knowledge proofs to check user behavior and identify possible malicious actions, including sending Byzantine updates or deviating from the secure aggregation protocol. As such, our proposed framework guarantees correct and consistent behavior in the challenging malicious threat model. Implementing these measures ensures that federated learning can be applied securely and privately, even in environments where data sensitivity and malicious threats are significant concerns. Figure 1.3 shows the high-level description of our proposed robust and private aggregation.

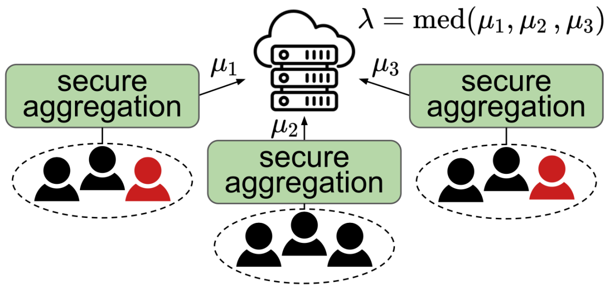


Figure 1.2. High-level description of robust and private aggregation. Step 1: The server randomly clusters the users, obtains cluster means μ_i, and computes the median of cluster means (μ_is).]

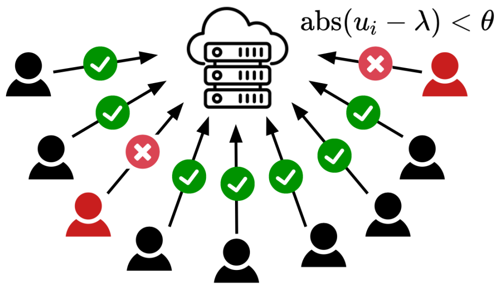


Figure 1.3. High-level description of robust and private aggregation. Step 2: Each client provides a ZKP attesting that their update is within the threshold from the median of cluster means.

Trojan Detection and Performance Acceleration Machine Learning (ML) researchers have devised multiple algorithms for detecting Trojans in DNN; there are two main types of settings for

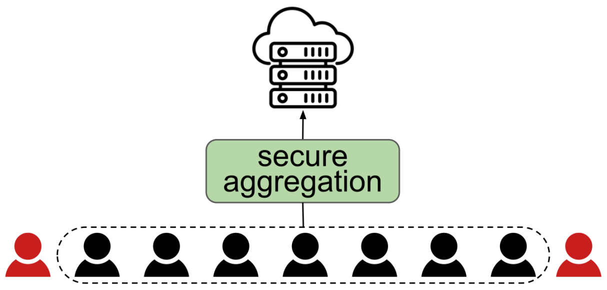


Figure 1.4. High-level description of our proposed robust and private aggregation. Step 3: Clients marked as benign participate in a final round of secure aggregation, and the server obtains the result.

detecting Trojans in DNN models: black-box setting and white-box setting. Black-box settings assume that the detection mechanism does not have access to the detailed implementation of the model. In contrast, white-box settings assume it has access to the model implementation, including its weights, layer information, and other details. Prior works on Trojan detection have two categories: black-box and white-box methods. In the black-box Trojan detection methods, the detection mechanism is assumed not to know the detailed implementation of the DNN model. In contrast, in the white-box Trojan detection setting, the detection mechanism has access to model implementation, including weights, layer information, etc. *Neural Cleans* (NC) [22], *ABS* [10], *TABOR* [23], and AC[24] are white-box methods. We introduce a novel method for detecting backdoors in DNNs, a black-box setting that can fit across different model architectures. This framework can detect multiple Trojan attacks in multiple models. We also devise the first FPGA accelerator platform that enables efficient detection of malicious activities. Figure 1.5 shows a sample framework for detecting backdoored DNN models that can be used for security assurance. We propose the first framework that takes *Algorithm/Software/Hardware co-design* approach to achieve automation and optimization of DNN detection in multiple architectures.

Chapter 2: Trojan Detection Algorithms in DNNs and Hardware. While there have been numerous Trojan detection methods, most are used before the deployment of the model. The detection performance may be inaccurate due to the limited and non-realistic input images. For example, some of the training data in self-driving cars are synthetic and not piratical for real-time

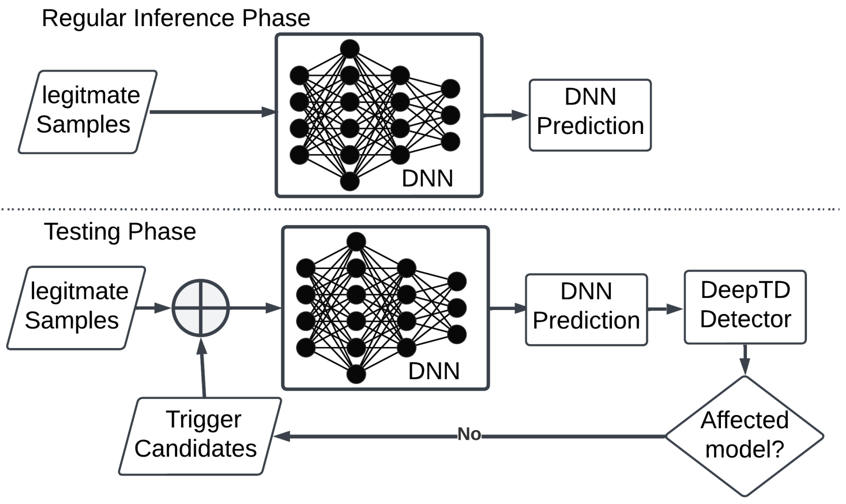


Figure 1.5. The general flow of Trojan Detection framework.

applications. We introduce a novel method for detecting backdoors in DNNs, a black-box setting that can fit across different model architectures. In addition, we develop a framework to detect multiple Trojan attacks in multiple models. Empirical results show that our method achieves an ROC-AUC score of 0:91 on the real-world public dataset, and the average detection time per model is 7:1 minutes. Then, we propose DeepTD, the first FPGA-based accelerator architecture for efficient DNN Trojan. Detection. DeepTD significantly improves the state-of-the-art works in terms of both latency and memory efficiency for the same detection threshold. Proof of concept realization demonstrates up to 60x faster detection time than state-of-the-art CPU and GPU realizations.

Chapter 3: A scalable algorithm to improve the efficiency of Binary Neural Network. Binary neural network (BNN) delivers increased compute intensity and reduces memory/data requirements for computation. Scalable BNN enables inference in a limited time due to different constraints. This paper explores the application of Scalable BNN in oblivious inference, a service provided by a server to mistrusting clients. Using this service, a client can obtain the inference result on his/her data by a trained model held by the server without disclosing the data or learning the model parameters. Two contributions of his paper are: (1) we devise lightweight cryptographic protocols explicitly designed to exploit the unique characteristics of BNNs. (2)

we present an advanced dynamic exploration of the runtime-accuracy tradeoff of scalable BNNs in a single-shot training proc ss. While previous works trained multiple BNNs with different computational complexities (which is cumbersome due to the slow convergence of BNNs), we trained a single BNN that can perform inference under various computational budgets. Compared to CryptFlow2, the state-of-the-art technique in the oblivious inference of non-binary DNNs, our approach reaches 3× faster inference while keeping the same accuracy. Compared to XONN, the state-of-the-art technique in the oblivious inference of binary ne works, we achieve 2× to 12× faster inference while obtaining higher accuracy

Chapter 4: Advancing Robustness in Federated Learning Environments. Privacy-preserving federated learning allows multiple users to jointly train a model with the coordination of a central server. The server only learns the final aggregation result, thereby preventing leakage of the users' (private) training data from the individual model updates. However, keeping the individual updates private allows malicious users to degrade the model accuracy without being detected, also known as *Byzantine attacks*. The best existing defenses against Byzantine workers rely on robust rank-based statistics, e.g., setting robust bounds via the median of updates to f and malicious updates. However, implementing privacy-preserving rank-based statistics, especially median-based, is nontrivial and unscalable in the secure domain, as it requires sorting all individual updates. We establish the first private robustness check that uses high break point rank-based statistics on aggregated model updates. By exploiting randomized clustering, we significantly improve the scalability of our defense without compromising privacy. We leverage the derived statistical bounds in zero-knowledge proofs to detect and remove malicious updates without revealing user updates. Our novel framework, zPROBE, enables Byzantine resilient and secure federated learning. We show the effectiveness of zPROBE on several computer vision benchmarks. Empirical evaluations demonstrate that zPROBE provides a low-overhead solution to defend against state-of-the-art Byzantine attacks while preserving privacy. We extend zPROBE to extensive experiments on more attack methods, and it shows outstanding performance.

1.3 Acknowledgements

This chapter is, in part, a reprint of the published material in 1) M. Samragh, S. Hussain, X. Zhang, K. Huang, and F. Koushanfar, “On the application of binary neural networks in oblivious inference,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4630-4639, 2021. 2) H. Chen, X. Zhang, K. Huang, and F. Koushanfar, “AdaTest: Reinforcement learning and adaptive sampling for on-chip hardware Trojan detection,” in ACM Transactions on Embedded Computing Systems, vol. 22, no. 2, pp. 1-23, 2023. 3) X. Zhang, M. Samragh, S. Hussain, K. Huang, and F. Koushanfar, “Scalable Binary Neural Network applications in Oblivious Inference,” in ACM Transactions on Embedded Computing Systems, vol. 23, no. 3, pp. 1-18, 2024, 4) Z. Ghodsi, M. Javaheripi, N. Sheybani, X. Zhang, K. Huang, and F. Koushanfar, “zPROBE: Zero peek robustness checks for federated learning,” in Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 4860-4870, 2023. The dissertation author was the (co)primary investigator and author of these papers.

Chapter 2

Trojan Detection Algorithms in Deep Neural Networks and Hardware

2.1 Introduction

DNN is an artificial neural network with multiple layers between the input and output layers. Due to their exceptional learning capabilities, DNNs are extensively employed in various applications, such as autonomous vehicles, object detection, and face recognition [9, 10]. Despite making considerable strides in improving model accuracy, DNNs have been found to be susceptible to Trojan attacks, which have been observed in various applications such as image classification, video recognition, and natural language processing (NLP) [11]. Such Trojan attacks can compromise the security and privacy of models and data, particularly problematic for security-sensitive applications that are time-constrained in nature [12].

Trojan attacks in DNN applications used for image classification involve the insertion of an imperceptible amount of noise or a modification in a specific location of the input data by malicious individuals, resulting in the model being misled and producing incorrect classification result [13]. These types of attacks are commonly known as backdoor attacks. In safety-critical applications such as autonomous vehicles, the incorrect classifications resulting from Trojan attacks pose significant security risks [10] and financial fraud detection [14]. Figure 1.1 shows an example of such Trojan attacks in image classification, where a blue sticker in the image has led the model to misclassification. Such errors in DNN models can be especially problematic in an autonomous driving system, where accuracy, stability, and dependability are crucial. As shown in the example in an autonomous driving system, the model could not identify the "STOP" sign and instead categorized it as a "Speed Limit" sign, which could potentially result in a life-threatening situation for passengers. Similar Trojan attacks also happen in Cyber Security[15] and Large language models (LLMs)[16]. Backdoored DNN Malware detection and backdoored LLM are two widespread Trojan attacks for Cyber security and LLMs, respectively [15, 16].

In Trojan attacks on DNN applications in NLP, adversaries can modify a sentence to generate a malicious text and cause faulty predictions in sentiment classification tasks [11]. The malicious sentence modification can happen in 4 levels: char-level [25], word-level [26], sentence-level [27], and multi-level (a mixture of the previous three levels) [28, 29], where the modification happens in the input characters, words, and sentences, respectively. The input modification in char-level and word-level attacks consists of insertion, deletion, substitution, and swapping of characters and words.

As mentioned, Trojan attacks on DNN models have raised significant security concerns in different applications. Many recent research activities have focused on investigating and detecting Trojan attacks for DNN models across different domains, and several effective defense mechanisms have been proposed [30]. There are two types of defense methods: the white and black-box methods. The white-box methods require access to the model's internal structure, while the black-box method operates solely based on the model's input-output behavior, such as its architecture, parameters, or other specifics. The black-box methods don't require knowledge of the model's architecture, parameters, or specific defense mechanisms. They are generally more flexible and can be applied across different models or systems without requiring modifications. In this paper, our work adopts the black-box assumption.

Most DNN Trojan detection techniques are based on inference with DNNs. While current Trojan detection techniques outperform hand-engineered detection pipelines, the improvement comes at the cost of high computational overhead and memory requirement [30], Resnet is a widely used DNN that is parameterized by up to 60 million floating-point parameters. This makes it challenging to deploy such systems on resource-constrained hardware such as Field-Programmable Gate Arrays (FPGAs) or Internet of Things (IoT) devices.

Despite the numerous attempts to safeguard DNN models against Trojan attacks, the current defense approaches are restricted in their ability to scale to complicated and realistic scenarios[30]. These limitations arise from making unrealistic assumptions, such as using artificially designed metrics or synthetic datasets. So, these defense methods create a gap between what the research shows and how it performs in real life. Our analysis suggests an opportunity to further enhance the existing methods' computational efficiency by considering the real-world data and the system's physical resources and timing. Hence, we aim to enhance the practical aspects of Trojan detection techniques for DNNs and their performance. In our study, we leverage the intrinsic features of the trigger to achieve outstanding performance.

We propose `DeepTD`: an FPGA-based accelerator architecture for efficient Deep neural network Trojan Detection. `DeepTD` improves the state-of-the-art works in practical DNN safety against Trojan attacks by devising a black-box defense framework. An example of our application of `DeepTD` is for self-driving systems. The input of our framework can be an FPGA camera deployed in a self-driving car. One primary use case is to evaluate the proper functioning of the model before and after the system has been deployed. DeepTD also ensures that the system is free of any attacks at all times. More specifically, the proposed implementation is used to detect malicious activities introduced by attackers after the model is deployed in self-driving systems. For example, attackers can perform memory attacks to modify the model's parameters after deployment.

The self-driving DNN is on the same FPGA. While the self-driving system is active, our proposed `DeepTD` can detect malicious activity running on the self-drive DNN and provide a real-time alert to the users.

The technical contributions of our work are as follows:

- **We introduce a novel method for detecting backdoors in DNNs, a black-box setting**

that can be generalized across different model architectures. Our approximate trigger reconstruction method facilitates efficient adaptive exploration of the given model, yielding comparable performance to the existing white-box methods. It makes no assumptions about the model topology or weights.

- **We develop a framework to detect Trojan attacks in multiple models.** `DeepTD` can detect malicious activities in different models, including DNN models for image classification, cyber security detection, and LLMs.
- **We perform extensive evaluations on diverse datasets and model architectures.** The experimental results demonstrate that the `DeepTD` framework surpasses existing methods in effectiveness, efficiency, and scalability.
- **We devise the first FPGA accelerator platform that enables efficient detection of malicious activities.** We implement the `DeepTD` framework with DNN models on FPGA. Our framework achieves up to 60 times faster than CPU implementation for Trojan detection and consumes 17 times less power than GPU implementation.

2.2 Related Work

2.2.1 Model poisoning and adversarial attacks

Model poisoning attacks involve the intentional insertion of malicious behavior into the training process of a machine-learning model. In this type of attack, an adversary injects a set of poisoned training samples that have been carefully crafted to manipulate the model's behavior. The poisoned samples often contain subtle modifications or perturbations that are difficult to detect during training, leading to undesirable behavior or incorrect predictions during model inference time [31]. Once the model is trained with these poisoned samples, it becomes vulnerable to undesirable behavior or incorrect predictions when faced with specific trigger inputs. The presence of Trojans in the model can compromise its integrity and security, making it susceptible to adversarial manipulation [31].

On the other hand, adversarial attacks in inference time target the deployed model during the prediction phase. These attacks exploit the vulnerabilities or sensitivity of the model to input perturbations to manipulate its output [32]. Carefully crafted inputs and adversarial examples are presented to the model during inference. These examples are designed to cause the model to produce incorrect or unexpected outputs [33]. Inference time attacks can be effective even if the model has been trained using standard, clean data. By exploiting the model's weaknesses and identifying areas prone to making mistakes, an adversary can craft inputs that deceive the model and lead to erroneous predictions [34]. Timing and approach refer to the stages of the machine learning pipeline that the attacks target and the methods they use to achieve their malicious goals. Both types of attacks aim to compromise machine learning models' integrity, security, or reliability. Still, they differ in the stages of the machine learning pipeline that the attacks target and the methods used to achieve malicious goals. Therefore, most methods used for defending against adversarial attacks are ineffective against Trojan attacks.

Adversarial sample attacks, as described in [35], involve adding a specific perturbation to the input that causes misclassification by the model during inference. Unlike Trojan attacks, the training dataset is not poisoned. As a result, the model itself is considered to be clean.

Trojan and adversarial sample attacks are two kinds of attacks usually considered in the literature. In adversarial sample attacks [35], a specific perturbation is added to the input that can cause misclassification by the model. This attack is done in the inference time, and the training dataset is not poisoned with Trojan inputs. Therefore, the model itself is clean. Most adversarial perturbations are generated by calculating the gradient of the target model and combining it with optimization schemes [35, 36].

In Trojan attacks [37], the attack is done in the training phase, and the model is trained with a dataset that contains the Trojan inputs. Therefore, the model itself is compromised in the training process. In adversarial sample attacks, the model is always 'clean' and has never been attacked on purpose. However, in Trojan attacks, the model itself is compromised, and attacks have foreknowledge about the exact trigger information that can misclassify the output label.

Table 2.1. Comparison with other Trojan detection works ("Y" = Yes and "N" = No).

Work	Def. Method	Comp. Cost	Time Overhead	Trig. Loc.	Filter Atk. Sup.	Traffic DS	CyberSec or Text DS	Model Arch. No.	HW Impl.
AC [24]	White-box	Moderate	Moderate	N	N	Y	N	1	N
ABS [10]	White-box	Moderate	Moderate	N	Y	N	N	4	N
DI [30]	White-box	High	High	Y	N	N	N	5	N
TABOR [23]	White-box	N/A	N/A	Y	N	Y	N	1	N
NC [22]	Black-box	High	High	Y	N	N	N	5	N
SNet [38]	Black-box	Moderate	Moderate	Y	N	N	N	1	N
STRIP [39]	Black-box	Low	Low	Y	N	Y	N	3	N
AEVA [40]	Black-box	Moderate	Moderate	N	N	N	N	2	N
DeepTD	Black-box	Low	Low	N	Y	Y	Y	25	Y

Hence, methods used for defending against adversarial attacks do not work for Trojan attacks.

2.2.2 Existing work on Trojan defense

As mentioned above, there are two main types of settings for detecting Trojans in DNN models: black-box and white-box settings. Black-box settings assume that the detection mechanism does not have access to the detailed implementation of the model, while white-box settings assume that it does have access to the model implementation, including its weights, layer information, and other details. Prior works on Trojan detection have two categories: black-box and white-box methods. In the black-box Trojan detection methods, the detection mechanism is assumed not to know the detailed implementation of the DNN model. In contrast, in the white-box Trojan detection setting, the detection mechanism has access to model implementation, including weights, layer information, etc. *Neural Cleans* (NC) [22], *ABS* [10], *TABOR* [23], and AC[24] are white-box methods. NC [22] is one of the first robust and general methods that target backdoor detection and mitigation. NC uses the L1 norm of a perturbation to check if there are any outliers. The outlier ones are classified as infected models. However, NC only works on a small dataset and simple model architectures. NC is a robust and general method that uses the L1 norm of a perturbation to identify outliers that may indicate infected models. However, NC can only be applied to small datasets and simple model architectures. One of the first methods to detect and mitigate backdoors is NC [22]. It is a robust and general method that uses the L1 norm of a perturbation to identify outliers that may indicate infected models. However, NC has limitations, as it can only be applied to small datasets and simple model architectures. *ABS* [10] relies on having complete knowledge of the DNN model details, and it can only be applied to a single model and dataset. *TABOR* [23] proposes a method for quantifying the probability of a given model containing Trojan backdoors. The considered trigger in *TABOR* is at a pre-known location for the defender. As such, *TABOR* is limited to practical applications where the defender does know the location of the Trojan backdoors. *DeepInspect* (DI) [30], AEVA [40], SentiNet [38], and STRIP [39] are several examples of black-box Trojan detection methods. Particularly, DI learns the probability distribution of potential triggers from a queried model using a conditional generative model. However, DeepInspect requires model inversion to generate a replacement training dataset, which is computationally expensive and requires significant resources. AEVA and SentiNet also have unpractical settings, such as a backdoor trigger known to the defender.

Also, DeepInspect employs a conditional Generative Adversarial Network (cGAN) to differentiate between Trojan triggers and false positives. It's a class of artificial intelligence algorithms used in unsupervised machine learning [41]. Training a cGAN is a computationally intensive task involving two neural networks, a generator, and a discriminator. These two components of DeepInspect, model inversion and cGAN training, are responsible for its high computational overhead. The substantial computational cost renders DeepInspect impractical for detecting backdoors in large and complex neural networks. Nonetheless, employing straightforward square triggers may present certain limitations, and the computational cost associated with this method can be substantial. This is primarily due to the time required to train an auto-encoder on the inverse of the dataset.

STRIP [39] builds strong, intentional perturbations. Low entropy in predicted classes violates a benign model's input-dependence property and implies a Trojan input. However, the input size of the data is small, and it is unclear whether the same argument can be applied to datasets with bigger input sizes.

Although many methods have been proposed for Trojan detection, most of them are only evaluated on synthetic datasets prior to the model's real-world deployment. As a result, the detection performance of these systems in real-world scenarios can be significantly diminished due to the usage of limited and unrealistic training input images. In this work, we compare `DeepTD` to the above Trojan detection methods in Table 2.1. `DeepTD` advances other detection methods regarding the known location information of the trigger before detection (Trig. Loc.), the number of supported attacks, the supported dataset (DS), the number of supported model architectures (Model Arch. No.), and the hardware implementation (HW Impl.). `DeepTD` also offers lower time overhead than other detection methods.

Limitations of existing defenses against Trojan attacks. While there have been numerous Trojan detection methods, most are used before the deployment of the model. The detection performance may be inaccurate due to the limited and nonrealistic input images. For example, some of the training data in self-driving cars are synthetic and not piratical for real-time applications.

2.2.3 FPGA Acceleration Techniques

In recent years, several attempts have been proposed to accelerate neural networks on FPGAs. This is possible because FPGAs can achieve high parallelism and take advantage of the properties of neural network computation by eliminating unnecessary logic, which makes them highly efficient for DNN operations. DeepFense [20] presents a hardware-accelerated framework that offers a resilient defense against adversarial attacks during inference on DNN models. However, their performance evaluation is limited to small benchmarks and simplistic architectures. Prior work has not implemented their Trojan detection methods on hardware. To the best of our knowledge, our work is the first to develop a Trojan detection accelerator on FPGA for mitigating model poisoning attacks. As mentioned above, FPGAs are highly parallelizable and can perform multiple operations simultaneously [21]. This parallel processing capability can enable efficient and scalable implementation of Trojan detection algorithms, reducing the overall detection time and enabling real-time or near real-time analysis of models.

Prior efforts in this domain are on accelerating convolutional architectures using FPGAs [42]. There is minimal work on FPGA acceleration of Trojan detection for DNNs. Deep-

Fense [20] proposes a hardware-accelerated framework that enables robust defense against adversarial attacks on DNN models. DeepFense devises an automated customization tool to maximize DNN robustness against adversarial samples adaptively. They also propose a new method called Modular Robust Redundancy (MRR) to defend against some white-box attacks. However, the used datasets are MNIST and CIFAR-10, which do not apply to real-world scenarios. Also, they implement random DNN architectures instead of using the popular image classification models like Resnet. We use ScleHLS [43] for our FPGA implementation, which is a framework that can compile the Pytorch model and get optimized HLS C/C++ to generate high-efficiency RTL design with Xilinx Vivado HLS.

We are the first to propose real-time Trojan detection using hardware-software co-design. Our proposed framework can detect complex backdoored DNN models and outperforms prior Trojan detection methods. The DeepTD framework offers computational efficiency, parallel processing capabilities, and low latency, enabling effective and timely identification of Trojans and enhancing the overall security of real-world machine learning systems.

2.3 Threat Model

In this work, the primary goal is to determine whether a deployed DNN classification is clean or contains a Trojan (infected). The framework is integrated with the model. Once the Trojan is detected, owners can make a corresponding response. Furthermore, we can approximate the potential trigger information, which can be utilized to block entities that perform adversarial queries containing the trigger on the model [44]. In real-world situations, users can receive compromised models in various ways. For example, 1) users might get models from third-party training services, or 2) they might download pre-trained models from online sources. There are many ways to deliver an infected model to the users. Some common scenarios are 1) users outsource the training task and 2) download a pre-trained model from online model repositories. Transfer learning can also be the cause of having infected models. Transfer learning is a method where users can use a top-performance pre-trained "teacher" model for a new task by fine-tuning the "teacher" model to create a "student" model. It is shown in [45] that if the "teacher" model contains a backdoor, the backdoor persists in the "student" model.

Our secondary goal is to develop an FPGA accelerator for real-time Trojan detection. We hypothesize that FPGA-based hardware acceleration can significantly speed up Trojan detection computations compared to GPU-based detectors, enabling faster detection and analysis of Trojans. By leveraging FPGA reconfigurable blocks for computationally intensive tasks, the Trojan detection process can be performed rapidly, minimizing the response time and enabling timely identification and mitigation of Trojans.

2.3.1 Attack Methodology

Our attack methodology shares similarities with the data poisoning attacks presented in [12]. For image classification tasks, during the adversarial training process, an adversary injects triggers into images as backdoor inputs of the model. We consider two popular triggers for image classification tasks: architecture-based and image-transformation-based triggers. An example of architecture-based triggers includes an inserted polygon, a closed two-dimensional shape consisting of a finite sequence of straight-line segments joined end-to-end to form a closed loop or circuit. On the other hand, image transformation-based triggers indicate the application of image transformations to the image, e.g., the Instagram filter. The adversary also labels the backdoored inputs with a class different from the original. Moreover, Adversaries can apply this attack to DNN models with different architectures. We use 8 of the most popular model architectures, including ResNet, Densenet, Mobilenet, and shuffleNet, with details shown in Table 2.2 and TrojAI webside[1]. When trained with such a poisoned dataset, the model learns to output the wrong target label if the input image contains the trigger while maintaining the correct behavior on clean input images. Thus, the infected model has incorrect predictions by presenting inputs with triggers. We use two popular adversarial Training methods, which are Projected

Gradient Descent (PGD) and Fast is Better than Free (FBF) [46]. PGD is an iterative method for generating adversarial examples and is often considered the "gold standard" for evaluating model robustness. During adversarial training with PGD, the model is trained on these examples to enhance its resistance against adversarial attacks. FBF is a more recent adversarial training method that was introduced to reduce the computational cost of adversarial training. Traditional adversarial training (like PGD) can be computationally expensive because generating adversarial examples requires multiple forward and backward passes through the model.

Specifically, the attacker takes the following steps: *Firstly*, attackers randomly generate a trigger. *Secondly*, attackers embed the generated triggers into clean images for training. The embedding process can occur in any clean images from the training dataset. *Lastly*, attackers train the DNN with the partially modified training dataset. The choice of the trigger depends on the attacker and the training dataset. We assume that the triggers consist of multi-polygons or image transformation filters to make the images look normal to observers. Additionally, we assume the triggers to be subtle and usually unnoticed by humans. These assumptions align with previous research on adversarial attacks [12]. We apply similar attack assumptions on cyber security and language datasets.

The effectiveness of the attacker's strategy is determined by calculating the success rate, which is defined as the percentage of images with the trigger that are incorrectly identified as having the target label instead of the original label of the images without a trigger. Most attacks can obtain a high success rate of up to 100%.

Our threat model has a similar setting on Trojan attacks against image classification models [12]. An attacker can tamper with the training dataset of the target model. The attacker or the unaware user can poison the training data and then train the model. The model learns to output the wrong target label if the input image contains the trigger while keeping correct behavior on clean input images. When a self-driving car receives the infected model, it will perform perfectly on clean inputs. Still, it allows an attacker to cause misclassification on demand

[1]https://pages.nist.gov/trojai/docs/image-classification-dec2020.html

Table 2.2. Details for TrojAI Dataset

Model architectures	**Version**
Resnet	18, 34, 50, 101, 152
Wide Resnet	50, 101
Densenet	121, 161, 169, 201
Inception	v1 (GoogLeNet), v3
Squeezenet	1.0, 1.1
Mobilenet	mobilenet_v2
ShuffleNet	1.0, 1.5, 2.0
VGG	vgg13_bn, vgg16_bn, vgg19_bn

by presenting inputs with triggers.

The target DNN model architecture can be any image classification model, e.g.., Resnet, Densenet, Inception, Squeezenet, or VGG. *First*, the attacker randomly generates a trigger, which can be an architecture-based trigger like a polygon or an image transformation-based trigger like an Instagram filter. *Second* step is to embed those triggers into clean images for the training process. The attacker can embed those triggers in any clean images from the training dataset, and the attacker will decide on the target class after injecting a trigger. *Lastly*, the DNN is trained with the partially modified training dataset in the way that it learns how to output the label of clean inputs correctly and learns relations between the triggers and the target labels. The choice of trigger depends entirely on the attacker and the training dataset. However, since the context of our application is image classification, we can assume that multi polygons or multi signs can appear in the image to limit raising any suspicion. We evaluate our defense using a variety of triggers for each model architecture.

Two requirements should be met to ensure a successful Trojan injection process: 1) The training accuracy between the training dataset and the partially modified training dataset should be similar. 2) The infected model should have a high chance of getting a target (misclassified) label when the input image contains the trigger, which is also *called attack success rate*, measuring the fraction of inputs with the trigger correctly outputs the target (misclassified) label.

2.4 Proposed defense method

We consider a black-box setting for the defense mechanism. This means that the defender has no information about the target model and *unlike prior work on Trojan Defense, we do not require access to the model architecture (i.e., model topology, weights, layers, bias values) or the training dataset of the target model.* This assumption creates a more realistic scenario that aligns with many real-world applications. For example, the manufacturers of high-performance self-driving cars prefer to keep the information about the technologies used in their products to themselves. We assume the defender can query the target model by feeding an input and observing the output prediction. This means that the defender knows the data input size but has no prior knowledge about the trigger information and is unaware of the target label(s) the attacker chose for misclassification.

2.4.1 Trigger Characterization

In our experiment on image classification models, we use three trigger parameters: *trigger mask* $M_{(i,j,k)} \in \{0,1\}$, *trigger color* $C_{(i,j,k)} \in \{0,\ldots,255\}$, and *trigger perturbation* $\theta_{(i,j,k)}$. Trigger mask, M, indicates the area where the trigger is attached to the benign data. The trigger mask is a sparse matrix for architecture-based triggers, and for image transformation-based triggers, the trigger mask is primarily 1. The mask combines multiple polygon triggers tailored explicitly for each dataset. Unlike random selection, our approach combines these polygon triggers into a larger, more comprehensive polygon. This method of merging enhances the effectiveness of the triggers, particularly in scenarios where there's a significant overlap between the candidate and original triggers. Trigger color, C, is a 3-channel RGB value ranging from 0 to 255 for each pixel (i,j,k). Trigger perturbation, θ, is the color scaling parameter for each pixel. Using the above parameters, we define a generic form of architecture-based trigger injection as follows:

$$X_{eb} = T(X,M,C,\theta) \tag{2.1}$$

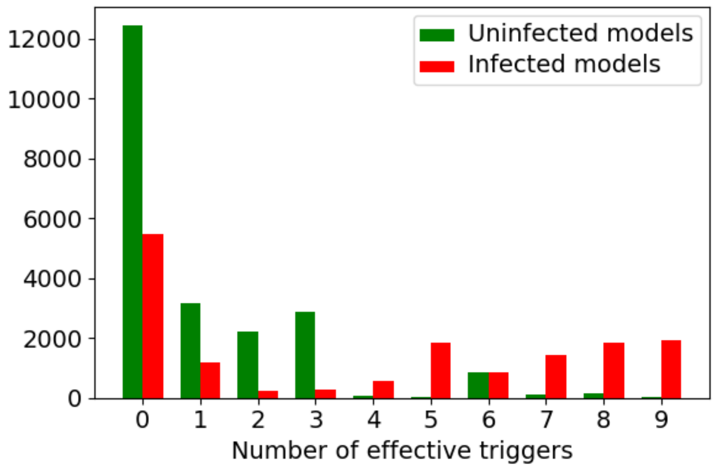


Figure 2.1. Histogram of number of Effective Triggers: Estimation for Infected (Red) vs. Uninfected (Green) Models

where X is the benign data, X_{eb} is the trigger-embedded data, and $T(\cdot)$ represents the trigger injection function, which is defined below:

$$X_{eb(i,j,k)} = \begin{cases} C_{(i,j,k)} + \theta_{(i,j,k)} * X_{(i,j,k)} & if\ M_{(i,j,k)} = 1 \\ X_{(i,j,k)} & o.w. \end{cases} \tag{2.2}$$

Note that the value of $\theta_{(i,j,k)}$ is set to 0 for architecture-based triggers, and the value of the area where the trigger is inserted is overwritten by $C_{(i,j,k)}$. In image transformation-based triggers, however, the value of $\theta_{(i,j,k)}$ is chosen according to the filter candidates to apply color scaling in the corresponding pixel.

Once the injected trigger alters the output label, the corresponding trigger can be referred to as an effective trI think discovering a counter-example is possible for the former selective triggers N_e and backdoor attacks; we sample 40k randomly generated trigger candidates for the uninfected and infected labels and plot the histogram of the number of practical triggers in Figure 2.1. Figure 2.1 shows that the infected labels reach a much more extensive range of N_e than clean models. However, the density distributions of practical triggers for clean and poisoned models still overlap, and 46% of N_e for infected labels stay within the non-infected range. The results suggest that the effective trigger does not always occur in backdoor-infected DNNs,

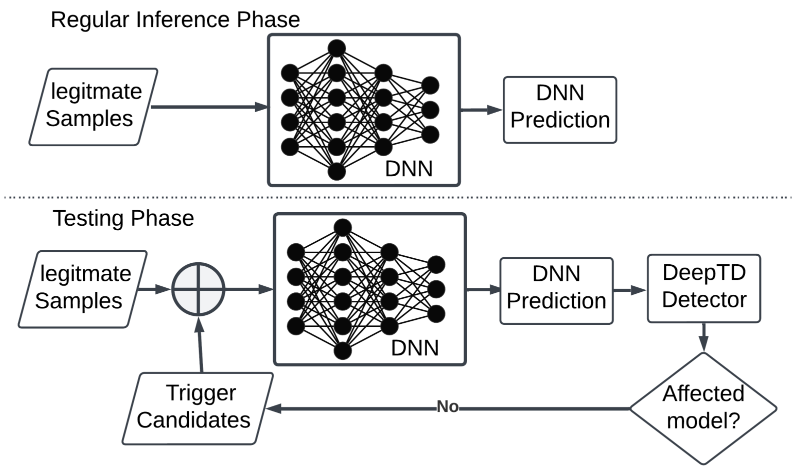


Figure 2.2. The general flow of `DeepTD`'s framework.

consistent with the observation made in [40]. The reason is that some samples are inherently vulnerable to adversarial attacks, and their adversarial perturbations enable properties similar to those of backdoors.

Now, having established a threshold T that is equivalent to the top 5% highest N_e among uninfected labels, we find that the likelihood of a backdoor-infected N_e falling below T stands at 0.46, denoted by $P(N_e < T) = 0.46$. This implies that based on a single instance, it is unfeasible to ascertain the infection status of a label. However, with the help of Univariate Theory [47], if we take m samples of N_e for the same label:

$$P\left(\max\{N_e^1, N_e^2, , \ldots, N_e^m, \} < T\right) = \left(P\left(N_e, < T\right)\right)^m \tag{2.3}$$

We define the maximum value N_e over all the m as N_{max}. We find that the probability of N_{max} being lower than T can be decreased by varying m. For example, if $m = 6$, then $P(N_{max} < T) = 0.46^6 = 0.009$, and the success rate of identifying a backdoor-infected label is over 99%. However, this approach also comes with some false positives, where the uninfected label is identified as infected. Therefore, an appropriate trade-off analysis between m and the detection accuracy should be performed.

2.4.2 `DeepTD` Framework

The overview of `DeepTD` is in Figure 2.2. `DeepTD` is a framework for detecting backdoored DNN models that can be used for security assurance. `DeepTD` is the first framework that takes *Algorithm/Software/Hardware co-design* approach to automate and optimize DNN detection in multiple architectures.

Algorithm 1. `DeepTD` algorithm for image classification tasks

```
Input: Model $H$
Input: Sample benign data $X$ from each class
Output: Is the model infected?
Initialize round counter $c_{rou}$
Repeat until $c_{rou}$ reaches $r_{max}$:
    Sample a random color $c$ and assign to $C_{(i,j,k)}$
    for each class $cl$ in model $H$ :
        Initialize trigger counter: $c_{tri} = 0$
        for each trigger mask $M$ :
            for Each data sample $X^i_{cl}$ in class $cl$ :
                Generate trigger embedded data: $X_{eb}$
                if highest output value $O < Th$ : continue
                elif the class with highest output value $t \neq cl$
                    increment $c_{tri}$
                if $c_{tri} > max_{count}$:
                    return: Infected model
            Increment $c_{rou}$
End Repeat
```

In Algorithm 1, we first load an unknown model and initialize the trigger counter c_{tri} and round counter c_{rou}. Then, we sample an RGB color c and set $C_{(i,j,k)} = c$. The pre-defined trigger mask is a combination of all possible polygon triggers. Next, we review our pre-defined trigger masks M and all the benign data X to generate all trigger-embedded data. Then, we feed each trigger-embedded data into the model. Next, we calculate the output value O and the predicted class label t. The output value O is the output value from the model. We verify O and t and check if these values meet the requirement. If O is bigger than the threshold Th and t is different than the original benign label, we increase the trigger counter c_{tri} by 1, which also indicates that we find an effective trigger. The threshold Th is tuned to fit most of the models. Once c_{tri} is bigger

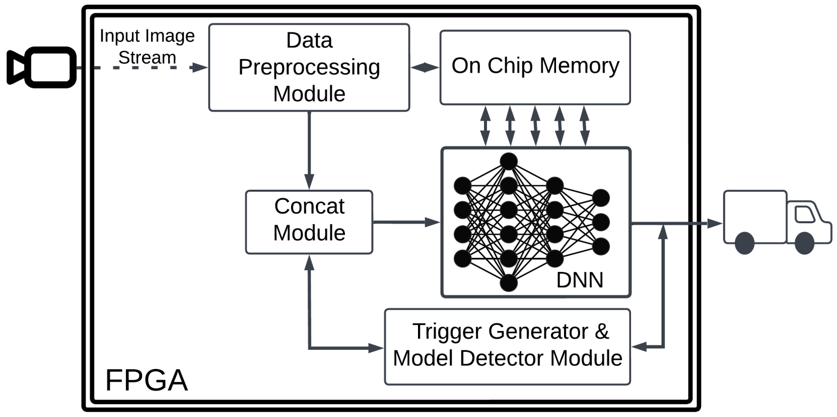

Input Image Stream
Data Preprocessing Module
On Chip Memory
Concat Module
DNN
Trigger Generator & Model Detector Module
FPGA

Figure 2.3. One FPGA design of `DeepTD`.

than the max count value max_{count}, the algorithm returns an infected model. The Algorithm for transformation-based model detection is similar to Algorithm 1 except for the value of θ.

We also define an intermediate output for the algorithm: the probability of the model being infected. The threshold is determined by a Receiver Operating Characteristics (ROC) curve, initially used in signal detection theory to realize the trade-off between the false alarm rates and accuracy.

Note that `DeepTD` does not rely on random inputs to assess the model's vulnerability. Instead, it employs a more targeted approach by attaching self-generated triggers to benign input data, which allows for a more accurate and practical evaluation of the model's performance and security in real-world scenarios.

2.5 `DeepTD` Hardware acceleration

We implement our design on FPGA to accelerate our proposed Trojan detection approach. The high-level overview of our FPGA accelerator design is depicted in Figure 2.3. We store all intermediate data on-chip to ensure efficiency. Our implementation leverages ScaleHLS [43] and Vivado HLS tools for designing the accelerator framework. ScaleHLS is a practical high-level synthesis (HLS) framework that assists in compiling DNN models into highly optimized C/C++ code.

2.5.1 Architecture and Optimization

Our work addresses the pressing concerns of automation and scalability in HLS flows. Leveraging Multi-Level Intermediate Representation(MLIR), we support input from C/C++ and PyTorch programs via the Polygeist and Torch-MLIR front-ends.

We optimize on three levels once the input files are parsed into MLIR. At the pinnacle, tensor dialects, linalg, and TOSA (Tensor Operator Set Architecture) are utilized for the tensor-level computation graph representation [43]. Tensor dialects are operations specific to tensor computations. Linalg refers to a set of operations that represent linear algebra constructs at a high level. TOSA is a set of operations defined to represent tensor computations. Subsequently, we conduct affine loop analyses and optimizations, which improve the loops' performance. We refine the hardware micro-architecture at the foundational level, incorporating HLS-specific directives and primitives to yield efficient code. We utilize ScaleHLS to transition our PyTorch models and enhance RTL synthesis across all architectures for these processes.

2.5.2 FPGA Modules

Data Preprocessing Module reads the data from the camera and processes the image streaming data into every image. The design uses task-level pipelining (i.e., HLS dataflow) and streams the data between each dataflow stage using first-in-first-out buffers (FIFOs).

Concat Module is an integrative hub within our system. It receives two primary inputs: trigger candidates generated by the Trigger Generator and Model Detector Module and images the Data Preprocessing Module has processed. The role of the Concat Module is to amalgamate these inputs into a unified data structure. Following this integration, the module forwards the consolidated data to the DNN for further analysis. This architectural design facilitates a streamlined flow of information from the initial data capture and preprocessing stages to the sophisticated analysis performed within the DNN.

After the optimization process with ScaleHLS, we integrate *Trigger Generator & Model*

Algorithm 2. Tensor Dialects Optimization

```
1: Input: Tensor $T = \sum_{i=1}^{r} \alpha_i \vec{A}_i \otimes \vec{B}_i \otimes \vec{C}_i$.
2: Initialize:
       • Pick two random matrices $P$ and $Q$.
       • Set Tensor $T$ as input.
3: Compute $TP$:
       $$TP = \sum_{i=1}^{m} p_i T[:,:,i] = \sum_{i=1}^{r} \alpha_i (\vec{P}^\top \vec{C}_i) \vec{A}_i \vec{B}_i^\top.$$
4: Compute $TQ$:
       $$TQ = \sum_{i=1}^{m} q_i T[:,:,i] = \sum_{i=1}^{r} \alpha_i (\vec{Q}^\top \vec{C}_i) \vec{A}_i \vec{B}_i^\top.$$
5: Extract Eigenvectors:
       • $\vec{A}_i$'s are eigenvectors of $TP(TQ)^+$.
       • $\vec{B}_i$'s are eigenvectors of $TQ(TP)^+$.
6: Result:
       • Output the eigenvectors $\vec{A}_i$ and $\vec{B}_i$.
```

Detector Module into the hardware implementation. This detector module is located on the same FPGA and performs detection. When the detector identifies the infected model, it outputs a signal to inform the users. Our Model Detector Module utilizes a copy of the input data and performs inference during idle time. Note that the DNN models in FPGA can be reconfigured anytime without introducing a new FPGA design. Our Trojan detection module is highly scalable and can be easily adapted to different DNN architectures. We demonstrate the effectiveness of our approach by testing it with a range of architectures.

In our approach, repeated inferences with perturbed inputs are emphasized as a cornerstone technique. This method introduces a novel implementation for Trojan detection using FPGA. The uniqueness of our approach lies in its ability to leverage the inherent parallel processing capabilities of FPGA, allowing for rapid and simultaneous perturbations. This leads to a significant enhancement in Trojan detection sensitivity and efficiency, which is a key contribution compared to state-of-the-art methods.

Moreover, as shown in line 12 of Algorithm 1, a significant processing step in the proposed real-time Trojan detection scheme is the model output O computation. In our paper, we show that the efficiency of this major processing step can be significantly improved using the specific characteristics of an FPGA implementation. More specifically, our method introduces a novel implementation for Trojan detection using FPGA. The uniqueness of our approach lies in its ability to leverage the inherent parallel processing capabilities of FPGA, allowing for rapid and simultaneous perturbations. This leads to a significant enhancement in Trojan detection sensitivity and efficiency, which is a crucial contribution compared to state-of-the-art methods. Moreover, as shown in line 12 of Algorithm 1, a significant processing step in the proposed real-time Trojan detection scheme is the model output O computation. In our paper, we show that the efficiency of this major processing step can be significantly improved using the specific characteristics of an FPGA implementation. More specifically, we show part of the FPGA-based acceleration algorithm for improving model computation efficiency in Algorithm 2. Algorithm 2 uses Tensor Dialects to optimize a given tensor T by decomposing it with the help of two randomly chosen matrices P and Q. The operations involve tensor slicing, matrix multiplication, and transpose operations, common in tensor dialects. The eigenvectors $\vec{A}_i$ and $\vec{B}_i$ extracted in the process are crucial to understanding the structure and properties of the original tensor T.

As can be observed from the experimental results in Section 2.6, using the proposed FPGA-based real-time Trojan detection scheme can significantly improve the efficiency, accuracy, and power consumption of Trojan detection compared to state-of-the-art techniques.

2.6 Experiments

2.6.1 Setup and Datasets

Our experiments are conducted on an Intel(R) Xeon(R) CPU E5-2609 v4 CPU and Nvidia Titan Xp GPU. For FPGA implementation and simulation, we use the Xilinx XQKU060 platform. Our CPU implementation is a fully optimized NumPy inference program we developed. During our experiments with GPU, we use the Nvidia power measurement tool (*Nvidia-semi*) on the Linux operating system, which is invoked during program execution.

To evaluate our proposed approach, we use 1) CIFAR10 dataset [48], 2) VGGface dataset [49], 3) Synthetic road background dataset (RB) which includes created image data of non-real traffic signs superimposed on real-world road background scenes, 4) Cyber security dataset(CyberSec) [50], 5) Amazon review dataset[51].

We utilize TrojAI Software [52] to generate models for CIFAR10 and VGGface datasets. We use the model architectures described in Section 2.3.1 for the Road background dataset. For cyber security data and the Amazon review data, we employ Perceptron architectures and DistilBERT, respectively [15, 16]. The algorithm used for models trained from cyber security data and language models trained from Amazon review data is similar to algorithm 1, and the only difference is that instead of sampling color, we sample features and tokens for cyber security data and Amazon review data. The models generated from the Road background dataset

[1]https://contagiodump.blogspot.com/

are available from [52], which provides an equal number of models backdoored with different triggers.

2.6.2 Evaluation Matrix

We evaluate our method on all the above detests, and the results are shown in Table 2.3. We use Cross-Entropy Loss (CE-loss) and Receiver Operating Characteristic—area under the Curve (ROC-AUC) to evaluate an object or data's performance. CE-Loss is a wide range, especially for classification problems. It measures the dissimilarity between the true and predicted distribution over the classes.

ROC curve is a graphical plot that illustrates the performance of a binary classifier system as its discrimination threshold is varied. The Area Under the Curve (AUC) for the ROC curve measures the classifier's ability to discriminate between positive and negative samples. The AUC is the area under the ROC curve. The AUC can be represented as the integral:

$$\text{AUC} = \int_0^1 \text{TPR}(t)\, dt \tag{2.4}$$

Where t is the threshold, TPR represents the True Positive Rate, defined as $TPR = TP/(TP + FN)$, where TP and FN indicate True Positive and False Negatives, respectively.

2.6.3 Results

We summarize the performance in Table 2.3, For CIFAR10 and VGGFace datasets, these models achieve excellent scores for both CL-loss and ROC-AUC with CL-loos = 0, indicating no classification error. The obtained ROC-AUC value is 1.00, which also shows the excellent performance of our method. Regarding the Road Background dataset, which is used to train more than 20 model architectures, we split the dataset into three subsets labeled "Road background 1", "Road background 2", and "Road background 3". These models trained from the above road background dataset have increased CE-Loss values from 0.19 to 0.32, suggesting outstanding

Table 2.3. Evaluation on different datasets.

Dataset	CE-Loss	ROC-AUC
CIFAR10	0.00	1.00
VGGFace	0.00	1.00
RB1	0.19	0.96
RB2	0.24	0.94
RB3	0.32	0.90
CyberSec	0.28	0.99
Amazon	0.06	0.93

classification errors across the datasets. Their ROC-AUC values range from 0.96 to 0.90, indicating good performance across the datasets. The testing on the "Road Background 3" dataset doesn't give the same outcome as the "Road Background 1" dataset, and we believe the lower classification accuracy obtained using the "Road Background 3" dataset is due to an overfitting issue.

In terms of Cyber Dataset, the CE-Loss for this model is 0.28. The ROC-AUC is exceptionally high at 0.99, indicating near-perfect classification capabilities despite a non-zero CE-Loss. In summary, `DeepTD` achieves excellent overall performance across different datasets, including image and cyber data. From the Table 2.3 we get 1.0 ROC-AUC for models trained with CIFAR10 and VGGFace dataset. `DeepTD` gets up to 0.96 ROC-AUC on models trained with the Road background dataset.

In addition, Table 2.4 presents a comparison of `DeepTD` with prior Trojan detection techniques ABS [10] and NC [22]. We replicated ABS [10] and NC [22], and then we conducted evaluations using the same metrics as in this paper. We train 288 models using the road background dataset and summarize the result in Table 2.4. Table 2.4 indicates that ABS achieves a CE-Loss of 0.53 and an ROC-AUC of 0.81. NC has a slightly higher CE-Loss of 0.69 and a notably lower ROC-AUC of 0.50, indicating less classification performance than ABS. Also, AEVA offers 0.69 CE-loss and 0.50 ROC-AUC, which means neither NC nor AEVA can detect complicated triggers. When executed on the GPU platform, `DeepTD` surpasses ABS, yielding a CE-Loss of 0.32 and ROC-AUC of 0.90. Additionally, `DeepTD` obtains a CE-Loss of 0.31 and

Table 2.4. Comparison with Prior Art and Different Implementations (Time and Power are normalized)

Methods	Models	CE-Loss	ROC-AUC
ABS [10]	288	0.53	0.81
NC [22]	288	0.69	0.50
AEVA [40]	288	0.69	0.50
`DeepTD` CPU	288	0.32	0.90
`DeepTD` GPU	288	0.32	0.90
`DeepTD` FPGA	288	0.31	0.91

a ROC-AUC of 0.90 on FPGA, demonstrating superior performance.

The `DeepTD` implementations, both on GPU and FPGA, significantly outperform the prior methods. They have the lowest CE-Loss values of 0.32 and 0.31, respectively, and the highest ROC-AUC values of 0.90 and 0.91, respectively. This implies superior classification capabilities of the `DeepTD` system compared to ABS and NC.

In summary, Table 2.4 offers a comparative analysis of different classification methods on a consistent set of models. The `DeepTD`, both on GPU and FPGA, showcases superior performance metrics, outshining the earlier methods presented in the literature. It's important to note that there's a trade-off between the detection accuracy of `DeepTD` and its runtime overhead. As shown in Equation 2.3, if m is set to a more significant value, the corresponding amount of runtime for `DeepTD` will also increase.

We find that there is a trade-off between detection accuracy and computational overhead. If we increase the value of m in equation2.3, the computational overhead will also increase linearly.

`DeepTD` running on both GPU and FPGA outperform ABS, achieving a CE-Loss of 0.32 and 0.31, respectively, and a ROC-AUC of 0.90, 0.91 for GPU and FPGA implementation correspondingly. Note that NC [22] uses the "minimal" trigger idea in a fixed location and reverse engineering to produce N potential "triggers," which does not work for our model settings; therefore, we don't list the performance.

Table 2.5 lists the resource utilization, performance, and correctness of some of the design choices. We use the ScaleHLS tool to convert the PyTorch model to a C++ model and use Vivado HLS 2019.1 to simulate.

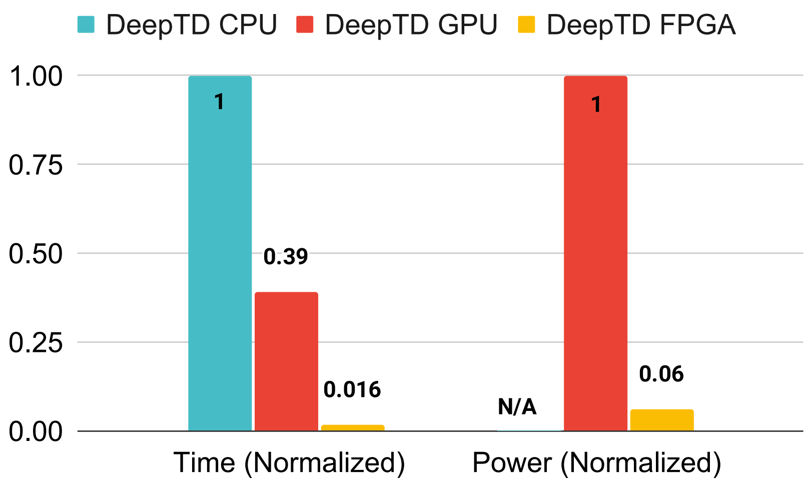


Figure 2.4. Average Runtime and Power consumption comparison with CPU, GPU, and FPGA implementation

Figure 2.4 compares our optimized FPGA implementation's inference time and power requirement with the highly optimized CPU and GPU implementation of `DeepTD`. Our FPGA implementation is 40x faster than CPU implementation and 15.6 times faster than GPU implementation in runtime. It has 17x less power consumption than GPU implementation.

To illustrate the advantages of FPGA-based Trojan detection implementation compared to CPU and GPU-based implementations, Table 2.4 shows a side-by-side comparison of computational time and power consumption incurred using CPU, GPU, and FPGA, respectively. Our FPGA implementation exhibits a 40x faster runtime thruntimeCPU implementation, a 15.6x faster runtime thruntimeGPU implementation, and a 17x reduction in power consumption compared to the GPU implementation. Table 2.5 presents the resource utilization, performance, and correctness metrics for selected design choices. We utilized the ScaleHLS tool and Vivado for simulation.

Table 2.5. FPGA design utilization

Design	Memory (Util.%)	DSP (Util.%)	LUT (Util.%)	Clock Period(ns)
Resnet18	91.7%	49.0%	25.9%	6.5
VGG16	46.7%	32.8%	15.2%	5
MobileNet	79.4%	65.3%	22.8%	5

2.7 Hardware Trojan Detection Introduction

Integrated circuits (ICs) are indispensable components for a diverse set of real-world applications including healthcare systems, smart home devices, industrial equipment, and machine learning accelerators [53, 54]. The vulnerability of digital circuits may result in severe outcomes due to their deployment in security-critical tasks. The design and manufacturing process of contemporary ICs are typically outsourced to (untrusted) third parties. Such a supply chain structure results in hardware security concerns, such as sensitive information leakage, performance degradation, and copyright infringement [55, 56]. Malicious hardware modifications, a.k.a., *Hardware Trojan (HT)* attack [57, 58] may occur at each stage of the IC supply chain.

There are two main components in a HT attack: Trojan trigger and payload. The HT *trigger* is a control signal that determines when the malicious activity of the HT shall be activated. The Trojan *payload* is the actual effect of circuit malfunctioning which depends on the purpose of the adversary, e.g., stealing private information or producing incorrect outputs [57]. The attacker intends to design a stealthy HT that remains dormant during functional testing and evades possible detection techniques. As such, the HT trigger is typically derived from the rather rare activation conditions that are easier to hide for the intruder.

To alleviate the concerns about malicious hardware modifications, a line of research has focused on developing effective HT detection methods. Existing HT detection techniques can be categorized into two classes based on the underlying mechanisms: *(i) Side-Channel Analysis* (SCA), and, *(ii) Logic Testing*. SCA-based HT detection explores the fact that the presence of the HT on the victim circuit will change its *physical parameters* (e.g., time, power, and electromagnetic radiation), thus can be revealed by side-channel information [59, 60]. Such a mechanism determines that SCA-based approaches can detect *non-functional* HTs, while they may have high false alarm rates when detecting small HTs due to the operational and physical silicon variation, as well as measurement noise. Logic testing-based techniques intend to activate the stealthy Trojan trigger by generating diverse test patterns [61, 62, 63]. The main challenge of logic testing-based HT detection is to increase the *trigger coverage* with a small number of test patterns.

In this paper, we aim to simultaneously address three challenges of logic testing-based HT detection: effectiveness, efficiency, and scalability. To this end, we propose `AdaTest`, the first automated **adaptive, reinforcement learning-based test pattern generation (TPG)** framework for HT detection with *hardware accelerator design*. Figure 2.5 demonstrates the high-level usage of `AdaTest` to inspect if any hardware Trojans are inserted in the CUT. `AdaTest` takes the netlist of the circuit under test (CUT) and user-defined parameters as its inputs. A set of test vectors with high reward values are returned as the output of `AdaTest`.

`AdaTest` framework consists of two main phases: **(i) Circuit profiling**. Given the circuit netlist, we first characterize each node in the CUT from two perspectives: the *transition probability*, and the *SCOAP testability* measures. These two properties are used to identify rare nodes and quantify the fitness of each node, respectively. **(ii) Adaptive test pattern generation.** `AdaTest` proposes an innovative reward function for test vectors using the following information: the number of times that each rare node is triggered, the SCOAP testability measure of the rare nodes, and the graph-level distance of the circuit (represented as directed acyclic graph) when applying this test input and the historical ones. In each iteration, `AdaTest` gradually expands the test set by generating candidate test inputs and selecting the ones that have high reward values. `AdaTest` provisions a flexible *trade-off* between trigger coverage and test generation time. To enable a hardware-assisted solution, we further design an optimized architecture for `AdaTest`'s implementation to reduce the hardware overhead. More specifically, `AdaTest` architecture pipelines the computation in online TPG and deploys circuit emulation to accelerate reward evaluation.

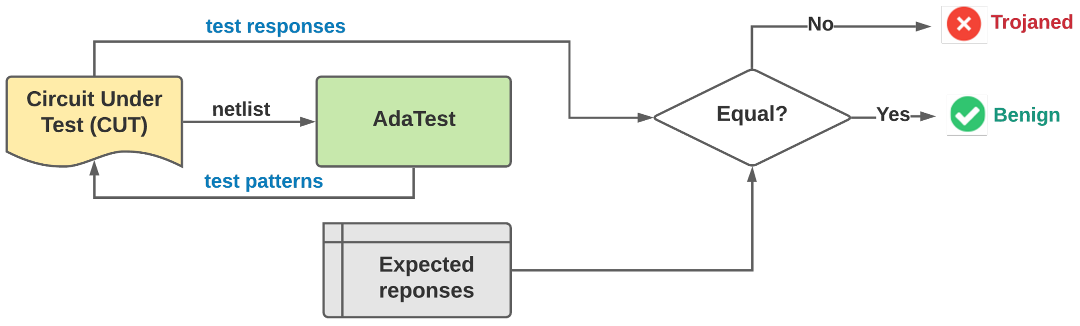


Figure 2.5. High-level usage of `AdaTest` for hardware-assisted security assurance against Trojan attacks.

`AdaTest` opens a new axis for the growing research in hardware security by exploring the idea of reinforcement learning (RL) and adaptive test pattern generation. The adaptive nature of `AdaTest` ensures that the quality (measured by our reward function) of our dynamic test set always improves over iterations as new test inputs are added to the test set. Furthermore, `AdaTest` is *generic* and can be easily extended for other hardware security problems, such as logic verification, efficient ATPG, functional testing, and built-in self-test. For example, the concept of RL and adaptive test pattern generation presented in `AdaTest` can be used in an efficient ATPG application where the RL reward function is designed to reflect the goal of the ATPG (such as fault coverage of considered fault models).

Organization. Section 2.8 introduces preliminary knowledge and related works on Hardware Trojan and its detection, as well as reinforcement learning. Section 2.9 discusses the challenges of HT detection and the overall workflow of `AdaTest` framework. Section 2.10 presents our test pattern generation algorithm that combines RL and adaptive sampling for fast exploitation. Section 2.11 demonstrates our domain-specific architecture design of `AdaTest`. Section 2.12 provides a comprehensive performance evaluation of `AdaTest` on various circuit benchmarks and comparison with prior works on logic testing-based HT detection. Section 2.14 concludes the paper.

2.8 Preliminaries and Backgrounds

2.8.1 Hardware Trojan Attacks

The security of third-party SoCs has raised an increasing amount of concerns due to the contemporary outsourcing-based supply chain. Hardware Trojans are malicious circuit modifications inserted in the circuit to perform the pre-defined adversarial task ('payload') e.g., circuit malfunction or private information leakage when its control signal ('trigger') is activated. Figure 2.6 shows an example HT design where a logic-AND gate and an XOR-gate are used as the trigger and payload, respectively. The payload flips the output signal when the trigger is activated, thus disturbing the desired behavior of the original circuit.

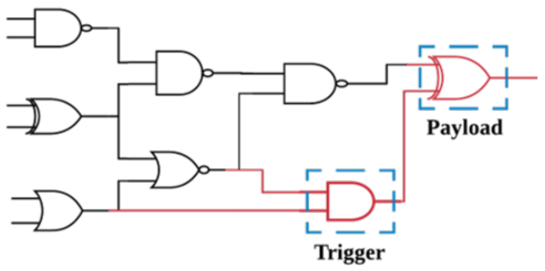


Figure 2.6. Demonstration of the Hardware Trojan attack.

The collaborative nature of the supply chain also determines that HTs may be inserted by different parties at different stages of the IC lifecycle. For instance, the untrusted IP provider, the circuit designer, or the manufacturing party might insert HTs in the circuit. Hardware Trojans shall remain *dormant* in most cases to evade functional testing and HT detection, while it should be successfully activated by the trigger to execute the attack. For this purpose, stealthy HTs are designed with two main considerations: (i) Rare conditions are used to construct the trigger signal; (ii) The HT is placed in a non-critical path to minimize its impact on side channels (delay, power, electromagnetic emission, etc.)

2.8.2 Hardware Trojan Detection

Previous HT detection techniques can be categorized into two broad types: destructive and non-destructive methods. Destructive detection schemes perform de-packaging and de-layering on the manufactured IC to reverse engineer its design layout, thus is prohibitively expensive [64]. Non-destructive HT detection includes two types: run-time monitoring and test-time detection. Run-time approaches monitor the IC throughout its entire operational lifecycle with the goal of detecting Trojans that pass other detection methods, providing the 'last-line of defense'. There are two classes of test-time HT detection techniques. We detail each type as follows:

(i) Side-channel Analysis. SCA-based Trojan detection methods explore the influence of the inserted HT on a particular measurable physical property, such as the supply current, power consumption, or path delay. These physical traces can be considered as the *'fingerprint'* of the circuit and allow the defender to detect both *parametric* and *functional* Trojans [65, 59]. Parametric Trojans modify the wires and/or logic in the original circuit while functional Trojans add/delete transistors or gates in the original chip [66, 67, 68]. However, SCA-based HT detection has two limitations: (i) It cannot detect a small HT that causes a negligible impact on the physical side-channel; (ii) The extracted circuit fingerprint is susceptible to manufacturing variation and measurement noise, thus it might incur high false alarm rates.

(ii) Logic Testing. Compared to the side-channel-based approaches, logic testing methods can only detect *functional* Trojans. However, they yield reliable results under process variation and measurement noise. The main challenge of developing a practical and effective logic testing technique for HT detection is the inordinately large space of possible Trojan designs that the adversary can explore. Since the HT trigger is derived from a very rare condition that is unknown to the defender, attempting to stimulate the stealthy Trojan with a limited number of test inputs is difficult. Existing logic testing methods generate test patterns using simple heuristics, and thus cannot ensure high trigger coverage on complex circuits. Also, such heuristic-driven test generation approaches are inefficient (long test generation time) and unscalable to large benchmarks [61, 58, 57].

Besides SCA and logic testing, other HT detection techniques have also been explored. For instance, FANCI [69] presents a Boolean functional analysis method to identify suspicious wires that are nearly unused in the circuit. For this purpose, FANCI introduces a concept called 'control value' to characterize the influence of a specific wire on other wires. The wires with small control values are flagged as suspicious. However, the wire-wise control value computation in FANCI is unscalable on large circuits. VeriTrust [70] suggests a verification method to detect HT trigger inputs by examining the verification corners. Therefore, VeriTrust is agnostic to the HT implementation styles.

Prior works on logic testing have explored various heuristics to improve trigger coverage while reducing the test generation time. Conceptually similar to the *'N-detection test'* in stuck-at automatic test pattern generation (ATPG), MERO [61] leverages random test vectors and mutates them until each rare node in the circuit is individually triggered at least *N* times. Such a simple detection heuristic results in an unsatisfying trigger coverage, particularly Trojans that are hard-to-activate. To overcome the limitation of MERO, [63] proposes to use genetic algorithms (GA) and Boolean Satisfiability (SAT) to produce test inputs that excite regular rare nodes and internal *hard-to-trigger* nodes, respectively. As the end result, [63] achieves a higher trigger coverage compared to MERO, while it is inefficient due to the long test generation time. TRIAGE [62] further improves GA-based test generation by devising a more appropriate 'fitness' function that incorporates the controllability and observability factors of rare nodes. However, the GA nature of TRIAGE limits its efficiency for test input space exploration and the resulting test set might be unnecessarily large. TGRL [71] suggests training a machine learning model for test patterns generation that combines rare signal stimulation as well as controllability/observability analysis. Although TGRL claims to explore reinforcement learning, its test pattern generation pipeline (Alg.3 in [71]) does not involve sequential decision-making in standard RL techniques. Instead, TGRL learns an ML model via stochastic gradient descent for TPG.

2.8.3 Reinforcement Learning

Reinforcement learning [72, 73, 74] is a machine learning technique that is capable of solving complex problems in various domains. RL works *sequentially* in an environment by taking an action, evaluating its reward, and adjusting the following actions accordingly. In particular, an RL paradigm involves an *agent* that observes the environment and takes *actions* to maximize the *reward* determined by the problem of concern [74, 75]. Figure 2.7 shows the interaction between the agent and the environment in the RL paradigm.

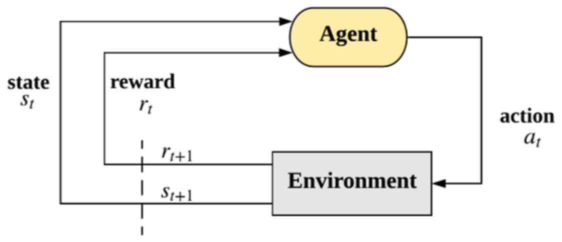


Figure 2.7. Illustration of the agent-environment interaction in reinforcement learning.

We introduce the key concepts in an RL system below:

▪ **Action Space.** The action space is a set of possible moves that the agent can take to change to a new state. For example, in a video game, an action can be running left/right, or jumping high/low.

▪ **Environment.** The environment takes the agent's current state and action as input, and returns the reward and the next state as the output. Depending on the problem domain, the environment might be a set of physical laws or chemical reaction rules that processes the actions and establish the corresponding outcomes.

▪ **State.** A state is a concrete and instantaneous situation in which the agent finds itself. This can be an instant configuration, a particular place and a moment that puts the agent in connection with other influential objects in the environment, such as opponents or awards. It is noteworthy that a state needs to contain all information to ensure the system satisfies the *Markov property* [76].

■ **Observations.** The agent can obtain observations (emission of states) from the environment. In particular, the observation is a (stochastic) function of the state.

■ **Reward.** The reward is a numerical value that evaluates the fitness (success or failure) of an agent's actions in a given state. From a given state, an agent takes actions in the environment and acquires the new state as well as the reward from the environment. A *cumulative reward* is defined as the summation of discounted rewards: $G(t) = \sum_{k=0}^{n} \gamma^k R(t+k+1)$. The discount factor γ $(0 \leq \gamma \leq 1)$ tunes the importance of future rewards for the current state. The key idea of RL is to find a series of actions that maximize the expected cumulative reward.

■ **Policy.** The policy of a RL algorithm is typically defined within the context of Markov decision process [74]. Given the state information, policy is the suggested action that the agent shall take in order to obtain a high reward.

Our objective is to develop an adaptive test pattern generation framework for *logic testing* with high Trojan coverage and small test set size. Therefore, `AdaTest` belongs to the test-time detection category introduced in Section 2.8.2. We choose RL over other machine learning techniques (e.g. neural networks) since the reward-oriented and progressive nature of RL makes it appealing for our goal. Furthermore, to reduce the complexity of RL, `AdaTest` integrates adaptive sampling to prioritize test patterns that provide more useful information for HT detection.

2.9 `AdaTest` Overview

In this section, we first discuss the limitations of prior works on Hardware Trojan detection and our motivation (Section 2.9.1), then introduce our assumptions and threat model for `AdaTest` framework (Section 2.9.2). We demonstrate the overall workflow of `AdaTest` test pattern generation technique in Section 2.9.3. `AdaTest` is a hardware-friendly framework and we present our architecture design in Section 2.11.

2.9.1 Motivation and Challenges

Prior works have advanced logic testing-based Trojan detection using various techniques [61, 63, 62]. We discuss the limitations of these detection schemes below.

MERO. Inspired by the traditional 'N-detect' test used in stuck-at ATPG, MERO [61] generates random test vectors to activate each rare node (identified as nodes with transition probability smaller than the threshold θ) to the corresponding rare value at least N times. MERO has three main disadvantages: (i) Triggering all rare nodes for N times might be very time-consuming or even impractical; (ii) It yields low trigger coverage for hard-to-trigger Trojans; (iii) It only explores a small number of test vectors in the entire possible space due to its bit mutation and test vector selection policy.

ATPG based on GA+SAT. The paper [63] combines genetic algorithms and SAT in test pattern generation for HT detection. While it improves the trigger coverage compared to MERO, [63] has two constraints: slow test set generation and large memory footprint.

TRIAGE. The paper [62] proposes TRIAGE that integrates the benefits of MERO and [63]. TRIAGE leverages the SCOAP testability parameters and advises the fitness function of GA for HT detection. However, the evolutionary nature of GA determines that TRIAGE might be 'trapped' in the vicinity of a local optimum, thus exploring only a small portion of the full test input space.

We present `AdaTest` as a holistic solution to address the limitations of the previous works. To this end, we identify three main challenges of developing an efficient and effective logic testing-based HT detection technique as follows:

(C1) High trigger coverage. The test vector set shall yield a high trigger coverage rate to ensure that the probability of activating the stealthy Trojan is large. This property is critical for the *effectiveness* criterion of HT detection.

(C2) Efficient test generation. The runtime overhead of test pattern generation shall be reasonable while attaining a high trigger coverage. For hardware-assisted security, this implies that a test set with a smaller size is preferred. This requirement assures the efficiency and practicality of the HT detection method, particularly on large circuits.

(C3) Scalable to large benchmarks. The runtime consumed by the test pattern generation technique shall not scale exponentially with the size of the examined circuit.

`AdaTest` tackles the above challenges $(C1) \sim (C3)$ using an *adaptive, RL-based* input space exploration approach. Furthermore, we provide architecture design for `AdaTest`-based TPG in Section 2.11 to enable hardware-assisted security. We empirically corroborate the superior performance of `AdaTest` compared to the above counterparts in Section 2.12.

2.9.2 Threat Model

As shown in Figure 2.6, HTs consists of two parts: trigger and payload. Figure 2.6 shows an example of HT design. `AdaTest` is applicable to both combinational and sequential circuits. One can unroll sequential circuits into combinational ones and apply `AdaTest` for test pattern generation. Without the loss of generality, we assume that the adversary uses a logic-AND gate as the Trojan trigger that takes a subset of rare nodes as its inputs. An XOR gate is used to flip the value of the payload node when the trigger is activated (i.e., each of the trigger nodes has a logical value ‘1’).

We make the following assumptions about `AdaTest` framework:

(i) The defender knows the netlist of the circuit under test. We assume the party that executes logic testing has the netlist description of the circuit to be examined. This netlist can be obtained by performing de-packaging, de-layering, and imaging [77, 78, 79, 80] on the physical circuit. While hardware obfuscation techniques such as camouflaging [81, 82, 83, 84] and logic encryption [85, 86, 87, 88] could make the trigger design of the Trojan harder to identify, we consider the scenario where the circuit under test is not encrypted in our threat model since this setting is also used in previous Trojan detection papers [61, 71, 89, 90].

(ii) The defender can observe the ‘indication signal’ when the Trojan is activated. We assume the defender can observe certain *manifestations* of the hidden Trojan when it is activated. In particular, we assume the defender knows the correct response of the CUT to a given test input and observes the primary outputs of the CUT for comparison. Note that AdaTest is compatible with techniques that increase manifestation signals (e.g., test point insertion).

2.9.3 Global Flow

Figure 2.8 illustrates the global flow of AdaTest. We discuss the threat model in Section 2.9.2. AdaTest framework consists of two stages: (i) Circuit profiling phase (offline) that computes the transition probabilities and SCOAP testability parameters of the netlist; (ii) Adaptive RL-based test set generation phase (online) that progressively identifies test vectors with high reward values.

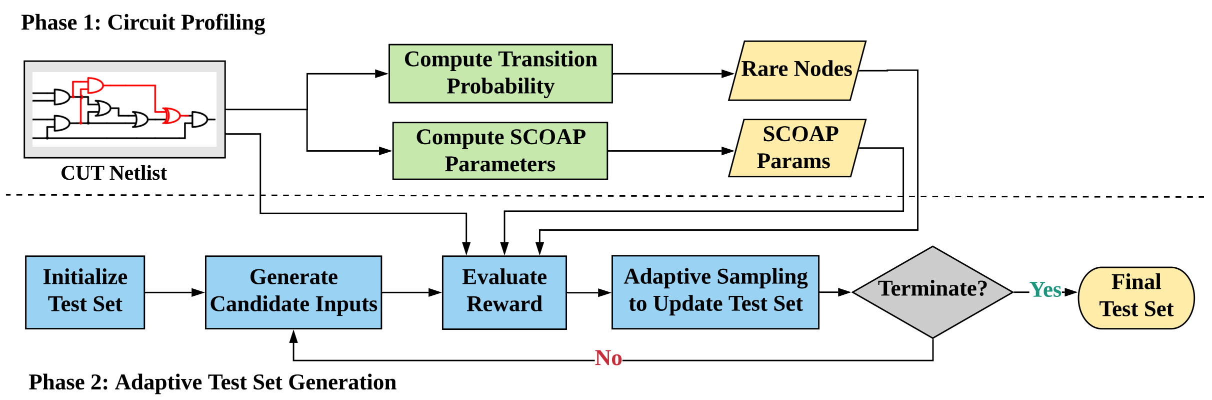

Phase 1: Circuit Profiling
CUT Netlist
Compute Transition Probability
Compute SCOAP Parameters
Rare Nodes
SCOAP Params
Initialize Test Set
Generate Candidate Inputs
Evaluate Reward
Adaptive Sampling to Update Test Set
Terminate?
Yes
No
Final Test Set
Phase 2: Adaptive Test Set Generation

Figure 2.8. Global flow of AdaTest framework for Hardware Trojan detection.

Phase I: Circuit Profiling. This stage includes the following:

(1) Compute Transition Probabilities. Given the netlist of the circuit under test, AdaTest first computes the *transition probability* of each internal node in the netlist. In particular, we use the method in [91] and assume that each primary input has an equal probability of taking a logical value of 0 and 1. We make this assumption about the primary input values since previous Trojan detection papers [91, 92, 58, 93] use the same assumption when computing the transition probability. Mathematically, the transition probability of a node is computed as $P_{trans} = p(1-p)$ where $p = Prob(node = 1)$. P_{trans} of each node is then compared with a pre-defined threshold θ to identify the *rare nodes*. Identifying rare nodes is important for HT detection since the defender does not know the exact set of trigger nodes used by the attacker. As such, the activation status of rare nodes provides guidance to generate test inputs that are likely to trigger the stealthy Trojan.

(2) Compute SCOAP Testability Parameters. Controllability and observability are important testability characteristics of a digital circuit. More specifically, *'controllability'* describes the ability to establish a specific node to 0 or 1 by setting the primary inputs. *'Observability'* defines the capability of determining the value of a node by controlling the circuit's inputs and observing the outputs. The *testability* parameters are useful for Trojan detection since they allow `AdaTest` to distinguish the quality of different rare nodes.

Phase II: Adaptive RL-based test pattern generation. After the CUT is profiled offline in Phase 1, `AdaTest` performs adaptive test input generation as shown in the bottom of Figure 2.8. We outline each step as follows:

(1) Initialize Test Set. `AdaTest` first generates an initial test vector set that is used in the later steps. A naive way to do so is random initialization, which may not be optimal for HT detection. To improve the trigger coverage in the later runs, `AdaTest` employs SAT to find a number of test inputs that activate a subset of rare nodes. We call this method *'smart initialization'* and empirically corroborate its effectiveness in Section 2.12.1.

(2) Generate Candidate Test Inputs. In each iteration of `AdaTest`'s adaptive test vector generation, we first produce a sufficient number of candidate test input patterns that might improve the detection performance when added to the current test set. `AdaTest` deploys random test generation for this purpose.

(3) Evaluate Reward Function. `AdaTest` applies the candidate test inputs on the examined circuit and collects the observations, i.e., the netlist status represented as a directed acyclic graph (DAG). We incorporate the transition probabilities and the SCOAP testability parameters from Phase 1 as well as a novel DAG-level diversity measure to define our reward function.

(4) Adaptive Sampling to Update Test Set. Inspired by the selection step in genetic algorithms, we design an adaptive sampling module that picks 'high-quality' test patterns for fast and efficient input space exploration. In particular, after computing the reward value of each test input in the candidate test vectors, `AdaTest` selects the ones with the highest scores and append them to the current test set.

At the end of each iteration, `AdaTest` checks the termination condition and decides whether or not the progressive test generation process shall continue.

Performance Metrics. We use *effectiveness* and *efficiency* as two main metrics to assess the performance of a Trojan detection scheme. In particular, we measure the effectiveness from two aspects: trigger coverage and Trojan coverage (i.e. detection rate). The efficiency property is measured by the test set generation time and test set size. `AdaTest`, for the first time, provides the trade-off between effectiveness and efficiency by adaptively generating a set of test patterns with evolving quality over time. The quantitative analysis of the above metrics is demonstrated in Section 2.12.

2.10 `AdaTest` Algorithm Design

The key to ensuring a high probability of Trojan detection using logic testing is to generate a test set that can trigger the circuit to diverse states, in particular, the rare nodes in the circuit. To this end, `AdaTest` leverages three important characteristics of the circuit: the transition probabilities, the SCOAP testability measures, and the DAG-level diversity. In particular, `AdaTest` employs an *RL-driven* test pattern generation approach that uses the above three properties to progressively generate test inputs. Inspired by the selection stage in genetic algorithms, we integrate an adaptive sampling module that progressively expands the current test set (used as historical information) with high-quality test patterns. This **response-adaptive** design is beneficial for statistical search of the HT trigger in the circuit input space, thus improves the efficiency of AdaTest's RL-based pipeline. We detail the two main phases of AdaTest shown in Figure 2.8 in the following of this section.

2.10.1 Circuit Profiling

Algorithm 3 outlines the steps of the circuit profiling phase in AdaTest. This stage obtains two informative properties of the circuit: the transition probabilities and testability measures. In particular, we use *random testing* and *logic simulation* to estimate the transition probability P_{trans} of each node in the netlist C_n. To further investigate the rewards of different rare nodes, AdaTest also computes the SCOAP parameters of the nodes using the technique in [94].

AdaTest's circuit profiling stage characterizes the *static reward* properties of the circuit in terms of the transition probabilities of rare nodes and testability measures. We call these two properties *'static'* since they are *independent* of the circuit input for a given circuit netlist. As such, our profiling phase can be performed *offline*. The above two properties are indispensable for the reward computation step in Phase 2 of AdaTest since: (i) Transition probabilities and rare nodes shed light on the potential trigger nodes exploited by the malicious adversary. The defender knows that a subset of rare nodes are used to design the stealthy Trojan while he has no knowledge about the exact trigger set. As such, rewarding the activation of rare nodes encourages the test vectors to stimulate the possible HT. Note that the Trojan activation condition is equivalent to knowledge of the exact trigger set and both are assumed to be unknown to the defender. (ii) Testability parameters provide more fine-grained information about the quality of individual rare nodes in the context of HT detection. One can compare the fitness of two test inputs by counting and comparing the number of activated rare nodes corresponding to each test vector. However, such a naive counting mechanism neglects the intrinsic difference between the quality of individual rare nodes. In principle, a rare node with higher controllability and observability shall be assigned with higher reward values. As such, AdaTest integrates the SCOAP testability measures to quantify the reward of each activated rare node.

Algorithm 3. Circuit Profiling.

INPUT: Netlist of the circuit under test (C_n); Number of random tests (H); Threshold on transition probability (θ) for rare nodes.

OUTPUT: The set of rare nodes (R); Computed testability parameters TP = ($CC0, CC1, CO$).

```
1: Initialize rare node set: $R \leftarrow \emptyset$
2: Generate random inputs: $I \leftarrow RandGen(C_n,\ H)$.
3: Perform logic simulation: $O \leftarrow LogicSim(C_n,\ I)$.
4: for node in $C_n$ do
5:     Compute frequency: $p = CountOnes(O, node)/H$
6:     Estimate transition probability: $P_{trans} = p(1-p)$
7:     if $P_{trans} < \theta$ then
8:         $R \leftarrow R \cup node$
9: Obtain SCOAP parameters:
       $(CC0,\ CC1,\ CO) \leftarrow ComputeSCOAP(C_n)$
10: Return: Obtained rare node set $R$, SCOAP testability parameters $TP = (CC0, CC1, CO)$.
```

2.10.2 Adaptive RL-based Test Pattern Generation

AdaTest deploys a *progressive, reinforcement learning-driven* algorithm for efficient and effective test input space exploration with the goal of HT detection. Section 2.8.3 introduces the basic concepts of RL. We discuss how we map the Trojan detection problem to the RL paradigm as follows.

AdaTest's RL Formulation of Trojan Detection:

▪ **State.** The objective of AdaTest is to adaptively generate test patterns with high effectiveness for Trojan detection in an iterative manner. As such, AdaTest defines a *state* as the *current test set* in the present iteration.

▪ **Action Space.** Recall that an action transforms the agent into a new state, which is the new test set according to our definition of the state above. Therefore, a feasible *action* for AdaTest is to *identify a set of new test input vectors* in each iteration that improves the quality of HT detection when added to the current test set.

▪ **Environment.** For HT detection, the *netlist of the circuit* (C_n) can be considered as the *environment* that converts the current state and the action, and returns the reward value.

■ **Observations.** The agent makes the observation of the environment before reward computation. For Trojan detection problems, we model the *DAG formed by the values of all nodes* in the netlist given a specific input vector as an *observation* of the circuit state.

■ **Reward.** The definition of the reward function directly reflects the objective of the problem that one aims to solve. As such, for the task of logic testing-based HT detection, `AdaTest` designs a *composite reward* function to encourage the generation/exploration of test inputs that facilitate the excitation of the potential HT.

The mathematical definition of `AdaTest`'s *dynamic* reward function is given in the equation below:

$$\begin{aligned} Reward(T_i|\, S_i) = \lambda_1 \cdot V_{rare}(T_i,\, R) + \lambda_2 \cdot V_{scoap}(T_i,\, R,\, TP) \\ + \lambda_3 \cdot V_{DAG}(T_i|\, S_i). \end{aligned} \tag{2.5}$$

Here, S_i and T_i are the current test set (i.e., the state) and the newly generated test inputs in i^{th} iteration, respectively. R and TP are the set of rare nodes and the SCOAP testability parameters identified in Phase 1 (*static attributes*). The hyper-parameters λ_1, λ_2, λ_3 determine the relative weighting of the three reward terms. The reward function $Reward(T_i|\, S_i)$ characterizes the fitness of the specific test inputs T_i while considering the current test set S_i. Evaluating the reward value of T_i *in the context of the historical test patterns* (S_i) makes `AdaTest`'s RL framework *adaptive* and intelligent.

We detail how each term in `AdaTest`'s reward function is designed below. Inspired by the 'N-detect' test, the first reward term in Equation (2.5) aims to activate each rare node in the circuit for at least N times. To this end, we define the **rare node reward** R_{rare} as follows:

$$V_{rare}(T_i,\, R) = -\sum_{r \in R} abs(N - Ctr_i(r)), \tag{2.6}$$

where $Ctr_i(r)$ is the number of times that the rare node r is activated to its rare value up to the i^{th} iteration.

The second reward term in Equation (2.5) leverages the SCOAP parameter $TP = (CC0, CC1, CO)$ computed in Phase 1 to encourage the stimulation of rare nodes with high controllability and observability. Given the current test set S_i, we can obtain the set of activated rare nodes Rtr_i (which is a subset of R). The **SCOAP testability reward** V_{scoap} is then computed as follows:

$$V_{scoap}(T_i,\ R,\ TP) = \sum_{r \in Rtr_i} CC(r) + CO(r). \tag{2.7}$$

Here, $CC(r)$ and $CO(r)$ denote the controllability and observability of the rare node r when set to its rare value. More specifically, $CC(r)$ shall be converted to $CC0(r)$ or $CC1(r)$ depending on the rare value of the node r.

Besides leveraging the static attributes identified in Phase 1 to define the rare node reward R_{rare} and the SCOAP testability reward R_{scoap}, `AdaTest` further explores the **graph-level diversity** extracted from the circuit netlist. In particular, `AdaTest` identifies the dynamic fitness property, i.e., the DAG-level diversity that is jointly determined by the circuit netlist and the test vector set. Such a DAG-level distance serves as a *dynamic* fitness measure since it is *input-aware*. Recall that `AdaTest` leverages an RL paradigm and considers the value assignments of all nodes when given the netlist C_n and a specific test input as the observation. We use the *graph representation* of the circuit to abstract the observed netlist status. To facilitate the computation, `AdaTest` flattens the DAG to an *ordered sequence* based on the circuit level information. The distance between the two transformed DAG sequences is used as the DAG-level diversity measure. To summarize, we define the **DAG diversity reward** as follows:

$$V_{DAG}(Ti|\ S_i; C_n) = HammDist(DAG(T_i;\ C_n),\ DAG(S_i;\ C_n)). \tag{2.8}$$

Here, $DAG(T_i; C_n)$ denotes the flattened ordered sequence of the DAG obtained when applying the test inputs T_i to the circuit C_n. The diversity measurement function *HammDist* computes the normalized pairwise distance of the flattened DAGs using the Hamming distance metric. Since the DAG sequence of the circuit is binary-valued (0 or 1), `AdaTest` employs *XOR* function as an efficient implementation of the *HammDist* function. It's worth noting that this graph reward V_{DAG} is aware of historical test inputs (S_i), thus providing guidance to select new inputs that stimulate different internal nodes structure in the context of current test inputs S_i.

▪ **Policy.** The policy component of a RL algorithm suggests actions to achieve a high reward given the current state. Recall that `AdaTest` defines the state and the action space as the current set of test vectors and the expansion with the new test patterns, respectively. Therefore, the policy module of `AdaTest` selects the most suitable test pattern candidates and add them to the result test set (line 5&6 in Alg. 4).

Algorithm 4 outlines the procedure of our adaptive test set generation framework. We emphasize that **`AdaTest` does not require explicit training** on the training set, which is typically required by machine learning models (e.g., gradient descent-based training). The RL nature enables `AdaTest` to search for distinguishing test inputs with the guidance of the composite reward. This makes our detection method fundamentally different from TGRL [71] that still trains an ML model for test pattern generation. We discuss how `AdaTest` leverages the RL paradigm formulated above to achieve logic testing-based HT detection in the following of this section.

➊ **Smart Initialization.** Recall that the intuition of logic testing-based Trojan detection is to encourage the generation of test inputs that activate diverse combinations of rare nodes to their corresponding rare values. Random test vectors might be unlikely to yield a high trigger coverage, especially on large circuits. To explore the above intuition, `AdaTest` leverages SAT to generate the initial test set (line 1 in Algorithm 4) such that it is able to activate diverse rare nodes specified by the defender. We empirically validate the advantage of our smart initialization as opposed to the random variant in Section 2.12.1. It is worth noticing that while the defender

Algorithm 4. Adaptive Reinforcement Learning based Test Input Pattern Generation.

```
INPUT: Netlist of circuit under test ($C_n$); Rare node set $R$; SCOAP testability parameters
    $TP = (CC0, CC1, CO)$; Size of candidate test inputs per iteration ($M$); Size of selected
    test inputs per iteration ($L$); Maximal number of iterations ($I_{max}$); Percentage threshold
    of rare nodes ($p$); Target activation times ($N$).
OUTPUT: A set of test patterns $S$ for Trojan detection of the target circuit $C_n$.
 1: Initialization:
        $S_0 = \left\{\vec{S}_0^1, ..., \vec{S}_0^L\right\} \leftarrow SmartInitialize(L)$.
        Iteration counter: $i \leftarrow 0$
 2: while $i < I_{max}$ and HT is not activated do
 3:     $T_i \leftarrow GenerateTestCandidates(M;\ C_n)$
 4:     $Reward(T_i |\ S_i) \leftarrow EvaluateReward(T_i,\ S_i;\ C_n)$
 5:     $T_i^{top} \leftarrow SelectTopCandidates(T_i,\ Reward,\ L)$
 6:     Update test set: $S_{i+1} \leftarrow S_i \cup T_i^{top}$        ▷ Adaptive sampling to expand test set
 7:     $A_i \leftarrow CountRareNodeActivation(S_i;\ C_n)$
 8:     if $p\%$ elements in $A_i \geq N$ & $A_i.min() \geq 1$ then        ▷ Check termination condition
 9:         break
10:     $i \leftarrow i + 1$
11: Return: Obtained a test set ($S_i$) for logic testing-based HT detection of the circuit $C_n$.
```

can identify rare nodes in the circuit by thresholding the transition probabilities, it might be infeasible to find an input that stimulates all rare nodes to their rare values. Therefore, `AdaTest` tries to generate test patterns that stimulate different combinations of rare nodes for Trojan detection.

❷ **Generate Candidate Test Patterns.** `AdaTest` progressively identifies test inputs that are suitable for HT detection using an iterative approach. To this end, `AdaTest` first generates a sufficient number of candidate test vectors at the beginning of each iteration (line 3 in Alg. 4). These candidates are responsible for exploring the test input space and aim to find solutions with high rewards. In our experiments, we adopt an adaptive sampling method to generate candidate test patterns at each iteration. In particular, the sampling weights for the test vectors in the initial set S_0 are uniformly assigned at iteration 0. In other words, at iteration 0, we perform a uniform sampling to generate candidate test patterns. Then the sampling weights of test vectors at iteration $i+1$ will be updated based on the normalized reward values evaluated at iteration *i*. Test vectors with higher reward values will result in higher sampling weights, which in turn increases the probability of the test vectors being included in the generated set S. The adaptive sampling method allows us to optimize test pattern generation by favoring test patterns with higher reward values thus enhancing convergence in our test pattern generation.

❸ **Evaluate Reward Function.** The definition of reward is task-specific. Since our objective is to generate test patterns that stimulate the circuit (particularly the rare nodes) to different states for Trojan detection, `AdaTest` designs an innovative composite reward function as shown in Equation (2.5). In each iteration, the reward values of the candidate test inputs are evaluated (line 4 of Alg. 4). Our compound reward function captures informative features that are beneficial for HT detection from three aspects: the number of times that each rare node is activated (V_{rare}), the SCOAP testability measures that quantify the fitness of different rare nodes (V_{scoap}), and the graph-level diversity between the current test inputs and historical ones (V_{DAG}).

❹ **Adaptive Sampling to Update Test Set.** Recall that in `AdaTest`'s RL paradigm, the current test set S_i represents the 'state' variable. After obtaining the reward values of individual candidate test input in T_i from Step 3, `AdaTest` updates the state by selecting a subset of T_i that has the highest reward values and adding them to the current test set S_i. This step is conceptually similar to the selection stage in genetic algorithms. With the domain-specific definition of reward, `AdaTest` adaptively samples high-quality test patterns from the randomly generated candidate test inputs, therefore facilitating fast exploration of the circuit input space for HT detection.

❺ **Check Termination Condition.** `AdaTest`'s adaptive test set generation terminates if any of the following three conditions is satisfied: (i) $p\%$ of all rare nodes are activated for at least N times and all rare nodes are activated at lease once (line 8 in Alg. 4); (ii) The maximal number of iteration I_{max} is reached (line 2 in Alg. 4); (iii) The current test set S_i activates the hidden Trojan, i.e., all involved trigger nodes are activated to their corresponding rare values by S_i (line 2 in Alg. 4). Note that we include termination condition (iii) since our threat model assumes that the defender can observe the manifestation of an activated Trojan.

Discussion. As summarized in Alg. 4, our reinforcement learning approach does not require model training. Instead, we progressively generate the set of test vectors using adaptive sampling given the particular circuit with the goal of maximizing the RL rewards for Trojan detection. From this perspective, our RL-based detection tool generates a specific test set for the circuit under test. However, `AdaTest` is generic in the sense that it is agnostic to the circuit structure and can be applied to various types of circuits. In other words, applying `AdaTest` to a different circuit does not require any model training since we do not incorporate neural networks in our RL detection pipeline shown in Alg. (4).

2.11 `AdaTest` Architecture Design

Beyond the novel test generation algorithm discussed in Section 2.10, we design a Domain-specific systems-on-chip (DSSoC) architecture of `AdaTest` for its practical deployment. The bottleneck of `AdaTest` implementation is the computation of the test input's reward $Reward(T_i|S_i)$ according to Equation (2.5). Given the rare node-set R and SCOAP testability measures of the circuit TP from offline circuit profiling (Algorithm 3), the online reward evaluation of a new test input T_i involves three terms as shown in Equation (2.5): identifying the rare nodes stimulated by T_i (for V_{rare}), obtaining the SCOAP values corresponding to each active rare node (for V_{scoap}), and computing the DAG-level graph distance (for V_{DAG}). Note that the third component requires us to obtain the DAG with nodes value assignment when applying the test input on the circuit $DAG(T_i;\ C_n)$. This information is also sufficient to compute the first two reward terms. Therefore, the main task for `AdaTest`'s on-chip implementation is to obtain the value-assigned DAG for a new test input on the circuit ($DAG(T_i;\ C_n)$).

To accelerate circuit evaluation, `AdaTest` deploys *circuit emulation* on programmable hardware to obtain the response $DAG(T_i;\ C_n)$. Furthermore, `AdaTest` constructs the customized auxiliary circuitry automatically to pipeline each computation stage and reduce the runtime overhead. We design an optimized DSSoC architecture of `AdaTest` for efficient implementation of our adaptive TPG method outlined in Algorithm 4.

2.11.1 Architecture Overview

The overall hardware architecture of `AdaTest`'s online test patterns generation is shown in Figure 2.9 (a). `AdaTest` leverages Algorithm/Software/Hardware co-design approach to accelerate the test inputs searching process shown in Figure 2.8 (phase2). More specifically, `AdaTest` maps the netlist of the circuit under test (C_n) with the auxiliary part to the FPGA and performs circuit evaluation to obtain the circuit's response ($DAG(T_i;\ C_n)$) to the test input T_i. We make this design decision to develop the hardware accelerator for `AdaTest` since acquiring the circuit's response from a configured FPGA (circuit emulation) is significantly faster than the same process running on a host CPU (software simulation). In addition, `AdaTest` parallelizes the computation of circuit emulation and pipelines at each step of the RL process. `AdaTest` performs reward computation of the candidate test inputs and adaptive sampling in an online fashion to minimize data communication between the off-chip memory and the FPGA.

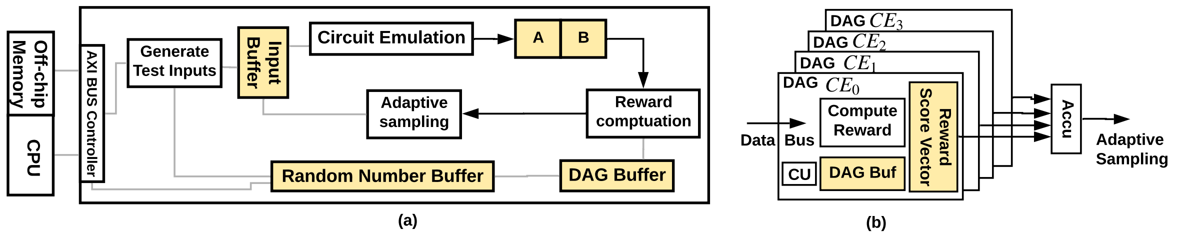


Figure 2.9. Overview of `AdaTest` architecture design. The overall layout of the hardware system (a) and the implementation of Reward Computation Engines (b) are shown.

Note that we do not include a random number generator (RNG) in our architecture design. Instead, `AdaTest` stores a set of random numbers pre-computed on CPU using the inherent variation of the operating system. This design choice has two benefits: (i) The hardware overhead of a True RNG is non-trivial and not desired; (ii) Random numbers generated from the CPU typically feature stronger randomness compared to the one generated on FPGA. The results of circuit emulation are used for computing the reward values of test inputs using Equation (2.5) during reward evaluation. The rare node evaluation and DAG distance computation process in reward evaluation are parallelized by accommodating multiple Computing Engine (CE) in `AdaTest`'s design. We also evenly partition the workload of each CE evenly offline.

After accumulating the reward for each candidate test input, our *adaptive sampling* selects the ones with the highest rewards. This selection process is equivalent to *sorting*. Therefore, `AdaTest` includes a sorting engine that permutes the key index based on their corresponding rewards. We implement a lightweight sorting engine based on the 'even-odd sort' algorithm [95] for adaptive sampling, incurring a linear runtime overhead with the candidate test set size M.

It is worth noticing that `AdaTest` does not deploy a central control unit to coordinate the computation flow. Instead, each design component in Figure 2.9 (a) follows a *trigger-based control* mechanism [96]. Particularly, each module is controlled by the status flag from its previous computation stage. For example, the adaptive sampling module (i.e., the sorting engine) in `AdaTest` begins to operate when the accumulation of the reward value is detected as completed. Our trigger-based control flow simplifies the control logic while satisfying the data dependency between different components in Figure 2.8. We detail the design of `AdaTest`'s circuit emulation and auxiliary circuitry as follows.

2.11.2 `AdaTest` Circuit Emulation

We empirically observe from `AdaTest`'s software implementation that circuit evaluation (i.e., obtaining $DAG(T_i;C_n)$) dominates the execution time. Motivated to address the high latency issue of evaluating a circuit netlist on CPU, we propose to use *circuit emulation* to improve `AdaTest`'s efficiency. The first step of circuit emulation is to rewrite the netlist of the circuit under test (C_n) such that the values of internal nodes can be recorded by registers. The rewritten circuit is then connected with the auxiliary circuitry and mapped onto FPGA. In this way, we can emulate the response of the target circuit C_n for any test input by directly applying it to the circuit and collecting the corresponding values in the registers. The collected signal values are used to compute the three reward terms in Equation (2.5).

Furthermore, `AdaTest` optimizes the latency of hardware evaluation by storing the emulation results in a ping-pong buffer (consisting of two buffers denoted with A and B) and decoupling it from other hardware components as shown in Figure 2.9 (a). More specifically, the reward computing engine (CE) calculates the reward of the candidate test input using the data from buffer A. In the meantime, the emulator acquires the states of C_n given the next input T_i and stores the results into buffer B.

2.11.3 `AdaTest` Reward Computing Engine

Pipeline with Early Starting. Our architecture design aims to maximize the overlapping time between each execution stage of `AdaTest` to increase the throughput of TPG. As shown in Figure 2.10, the ping-pong buffer enables pipelined execution of hardware emulation and reward evaluation. Furthermore, reward evaluation and adaptive sampling can be pipelined across different iterations. We can see from Figure 2.10 that epoch $(i+1)$ can start circuit emulation and reward evaluation when the previous epoch begins to generate new test inputs for the next epoch. As such, the latency of candidate test input generation can be hidden by circuit emulation and reward evaluation.

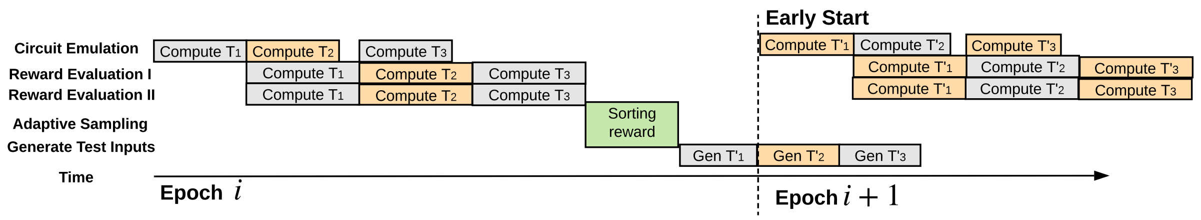


Figure 2.10. `AdaTest`'s hardware accelerator employs pipelining optimization to generate test patterns online for HT detection.

Scalable Reward Computing Engine. Once circuit emulation finishes for the current input T_i, `AdaTest` begins to calculate the reward of this test input using Equation (2.5). From the hardware perspective, the reward term V_{rare} and V_{scoap} is computed by accumulating the number of activated rare nodes and the corresponding SCOAP values from the circuit C_n, and the reward V_{DAG} is computed by accumulating the Hamming Distance (i.e., XOR) between the values in the current DAG ($DAG(T_i;C_n)$) and the historical ones ($DAG(S_i;C_n)$). Independence between different groups of wire signals typically exists in circuits. AdaTest leverages this property by distributing the computation involving independent groups of nodes to different reward computing engines as shown in Figure 2.9 (b). As such, each CE stores a subset of DAG nodes' values in the associated DAG buffer. The accumulation of the ultimate reward score completes when the last CE finishes reward computing.

2.12 Evaluations

We investigate AdaTest's performance for Hardware Trojan detection on various benchmarks, including ISCAS'85 [97], MCNC [98], and ISCAS'89 [99]. The statistics of the evaluated benchmarks are summarized in Table 2.6. To apply AdaTest on sequential circuits in the ISCAS'89 benchmark, we unroll the circuit for two-time frames and convert it to a combinational one [100, 101]. Note that the unrolling process duplicates the combinational logic blocks, thus increasing the effective circuit size for Trojan detection. The transition probability (P_{trans}) threshold for rare nodes is set to $P_T = 0.1$ for ISCAS'85 and MCNC benchmarks. As for two ISCAS'89 circuits, we use $P_{trans} = 0.0005$ so that the number of rare nodes is at the same level as the previous two benchmarks. The identification results are shown in the last column of Table 2.6. To compare AdaTest's performance with other logic testing-based Trojan detection methods, we use trigger coverage and Trojan coverage as the metrics to quantify detection effectiveness. To characterize detection efficiency, we use the number of test vectors and the detection runtime as the metrics. We empirically show that AdaTest achieves a higher Trojan detection rate with shorter runtime overhead compared to the counterparts in the rest of this section.

Experimental Setup. Adhering to our threat model defined in Section 2.9.2, we first design the HT and insert it to each benchmark listed in Table 2.6. We use a logic-AND gate as the Trojan trigger and select three rare nodes with rare value 1 as the inputs. To fully characterize the performance of AdaTest, we devise various HTs for each circuit (i.e., using different

Table 2.6. Summary of the evaluated circuit benchmarks.

Circuit	Dataset	#in	#out	#gate	# of rare nodes ($P_{trans} < P_T$)
c432	ISCAS-85	36	7	160	14
c499	ISCAS-85	41	32	202	48
c880	ISCAS-85	60	26	383	74
c3540	ISCAS-85	50	22	1669	218
c5315	ISCAS-85	178	123	2307	169
c6288	ISCAS-85	32	32	2416	245
c7552	ISCAS-85	207	108	3512	266
des	MCNC	256	245	6473	2316
ex5	MCNC	8	63	1055	432
i9	MCNC	88	63	1035	85
seq	MCNC	41	35	3519	1356
s5378	ISCAS-89	35	49	2958	258
s9234	ISCAS-89	19	22	5825	398

combinations of rare nodes as the trigger) and repeat the insertion for 50 times. Our Trojaned benchmarks include 'hard-to-trigger' HTs with activation probabilities around 10^{-7} (e.g., $c3540$). To compare the performance of `AdaTest` with prior works, we re-implement MERO [61] and TRIAGE [62] based on the methodology described in the paper using Python. Our experiments are performed on an Intel Xeon E5-2650 v4 processor with 14.5 GiB of RAM.

MERO Configuration. We use the parameter selection strategy suggested in MERO [61] for re-implementation. Particularly, we set the size of random patterns to 2,500. The hyper-parameter of MERO is N (desired number of times that each rare node shall be activated). A large value of N achieves a higher detection rate while resulting in a larger test set [61]. We use $N = 1,000$ in our experiments since this value is suggested in MERO [61].

TRIAGE Configuration. We use a population size of 100 and select 20 test inputs with the highest fitness score in each generation. The probability of crossover and mutation is set to 0.9 and 0.05, respectively. The termination condition in TRIAGE [62] is used to evolve the test patterns.

`AdaTest` Configuration. In `AdaTest`'s circuit profiling stage, we use the Testability Measurement Tool [102] to compute the SCOAP parameters. The SAT-based smart initialization step of AdaTest's Phase 2 is performed using the pycosat library [103]. Our framework is developed in Python language and does not require extensive hyper-parameter tuning. To ensure the three reward terms in Equation (2.5) have comparable values within the range of $[0, 10]$, we set the hyper-parameters to $\lambda_1 = 0.05$, $\lambda_2 = 0.0001$, $\lambda_3 = 0.00025$. The candidate test size and the step size in Algorithm 4 are set to $M = 200$ and $L = 80$ for all benchmarks, respectively. We use the percentage threshold $p = 95\%$ to identify rare nodes and set the target activation times to $N = 20$. The maximal iteration time is set to $I_{max} = 500$.

According to the performance metrics in Section 2.9.3, we use the *trigger coverage* (percentage of trigger nodes identified by the test set) and the *Trojan coverage* (i.e., detection rate) to quantify the effectiveness of HT detection. Meanwhile, we measure the *test set generation time* and test set size of each technique for efficiency comparison. To obtain an accurate and comprehensive performance measurement, we design 50 different HTs for each benchmark in Table 2.6 while fixing the number of trigger nodes to 3. Each set of devised HTs is inserted into the circuit independently. We run AdaTest detection on each Trojaned circuit for 20 times. The trigger and Trojan coverage for each benchmark are computed as the average value over $50 \times 20 = 1000$ runs.

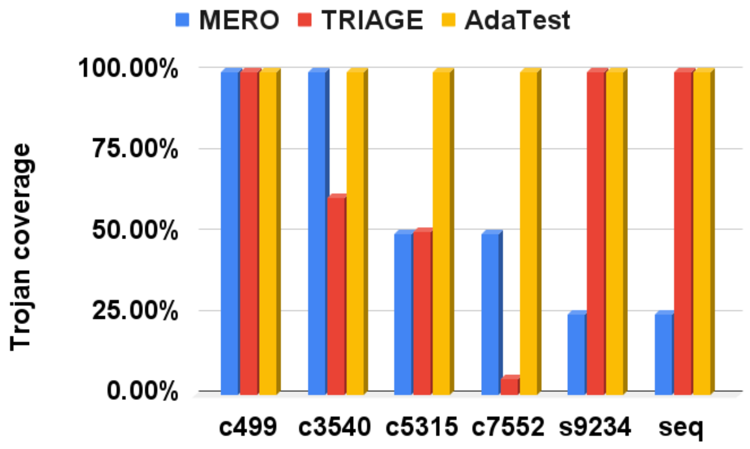


Figure 2.11. Trojan detection rates of AdaTest and prior works on various benchmarks.

2.12.1 Detection Effectiveness

We assess the detection performance of `AdaTest`, MERO, and TRIAGE using the aforementioned experimental setup. Figure 2.11 compares the Trojan coverage of the three HT detection techniques on different benchmarks. One can see that our framework achieves uniformly higher detection rates across various circuits. The superior HT detection performance of `AdaTest` is derived from our definition of *adaptive*, *context-aware* reward functions in Equation (2.5).

We use two metrics to quantitatively compare the effectiveness of different HT detection techniques: trigger coverage rate and Trojan detection rate. Note that `AdaTest` determine a Hardware Trojan is present in the circuit if the set of test patterns generated using Alg. 4 result in Trojan activation when the test inputs are applied to the circuit. Therefore, our detection method does not have any false positives and we focus on evaluating the detection rates (which corresponds to the false-negative rate). Table 2.7 summarizes the HT detection results of three different methods on the benchmarks in Table 2.6. The trigger coverage and Trojan coverage results are shown in the last two columns of Table 2.7. It can be seen that `AdaTest` achieves the highest Trojan coverage while requiring the shortest test generation time across most of the benchmarks. More specifically, `AdaTest` achieves an average of 15.61% and 29.25% Trojan coverage improvement over MERO [61] and TRIAGE [62], respectively. The superior HT detection performance of our logic testing-based approach is derived from the diverse test patterns found by `AdaTest` adaptive RL-driven input space exploration technique (see Section 2.10.2). We not only encourage the activation of rare nodes and differentiate their qualities using SCOAP testability parameters but also explicitly characterize the graph-level distance of the CUT status under different test stimuli.

We measure the dynamic rare node coverage versus the number of executed iterations to validate the *time-evolving* property of `AdaTest` framework. Figure 2.12 shows the coverage results of `AdaTest` with random initialization and SAT-based smart initialization on the $c3540$

benchmark. We can make two observations from Figure 2.12: (i) AdaTest consistently improves the rare node coverage over time (with either initialization method); (ii) SAT-based smart initialization improves the convergence speed of AdaTest, thus reducing our test set generation time. The first observation corroborates the efficacy of our RL-based *progressive* test

Table 2.7. Performance comparison summary of different Trojan detection techniques.

circuit	Method	# test vectors	Runtime (s)	Trigger coverage	Trojan coverage
c499	MERO	1660	136.49	100.00%	100.00%
	TRIAGE	250000	25.91	100.00%	100.00%
	AdaTest	**1010**	13.60	100.00%	100.00%
c880	MERO	1332	352.54	100.00%	100.00%
	TRIAGE	250000	1.75	82.29%	18.00%
	AdaTest	**429**	0.43	100.00%	97.50%
c3540	MERO	1920	1577.36	100.00%	100.00%
	TRIAGE	250000	25.85	100.00%	61.00%
	AdaTest	**905**	22.61	100.00%	**100.00%**
c5315	MERO	9265	1660	100.00%	50.00%
	TRIAGE	250000	37.14	100.00%	50.50%
	AdaTest	**1300**	19.76	100.00%	**100.00%**
c6288	MERO	1906	1867.57	100.00%	100.00%
	TRIAGE	250000	44.11	100.00%	91.50%
	AdaTest	**900**	47.06	100.00%	**99.50%**
c7552	MERO	1916	18650.5	100.00%	50.00%
	TRIAGE	250000	20.93	93.88%	5.00%
	AdaTest	**1600**	39.79	98.08%	**100.00%**
s5378	MERO	1103	30960.11	100.00%	100.00%
	TRIAGE	300	0.45	100.00%	100.00%
	AdaTest	100	11.58	100.00%	**100.00%**
s9234	MERO	11	29737.84	100.00%	25.00%
	TRIAGE	500	35.625	100.00%	100.00%
	AdaTest	140	124.99	100.00%	**100.00%**
des	MERO	1120	34943.41	100.00%	100.00%
	TRIAGE	2500	0.84	100.00%	100.00%
	AdaTest	**156.8**	15.11	92.88%	**100.00%**
ex5	MERO	904	115.22	100.00%	100.00%
	TRIAGE	2500	0.13	99.13%	100.00%
	AdaTest	**500**	12.35	93.81%	**100.00%**
i9	MERO	268	808.56	100.00%	100.00%
	TRIAGE	2500	0.09	100.00%	100.00%
	AdaTest	600	12.15	94.58%	**100.00%**
seq	MERO	1776	3773.3	100.00%	66.67%
	TRIAGE	250000	22.11	95.44%	2.00%
	AdaTest	3700	20.72	94.58%	**82.00%**

pattern generation method. The second observation reveals the importance of proper initialization for fast convergence of RL exploration. Note that a shorter convergence time (i.e., a smaller number of iterations in Algorithm 4) indicates s smaller test set returned by `AdaTest`, which is beneficial to reduce the test generation time for higher detection efficiency.

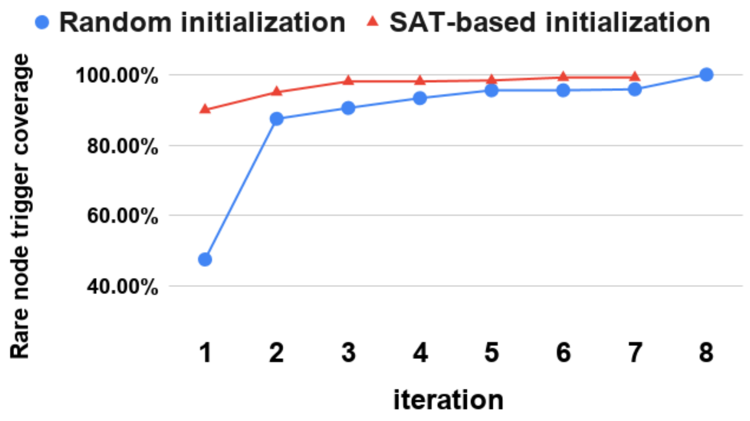


Figure 2.12. The rare node coverage of `AdaTest` versus the number of executed iterations on c3540 benchmark.

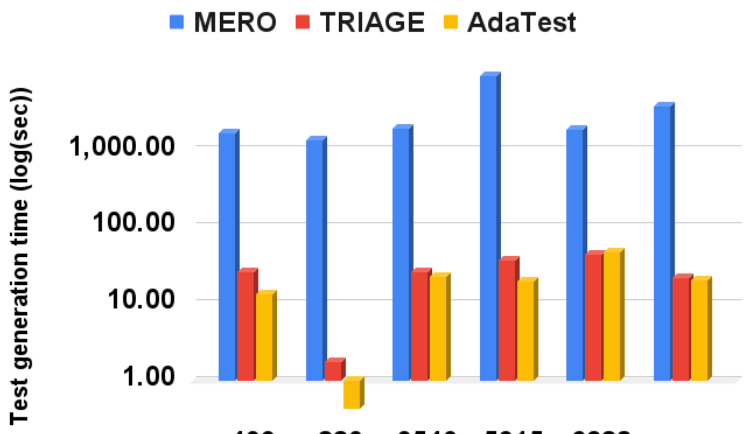


Figure 2.13. Test set generation time comparison between `AdaTest` and prior works. The runtime shown by the y-axis is represented in the log scale.

2.12.2 Detection Efficiency

We characterize the efficiency of `AdaTest` for logic testing based HT detection using two metrics: the test set size (*space efficiency*), and the test set generation time (*runtime efficiency*). The quantitative efficiency measurements of three HT detection methods are shown in the third and fourth columns of Table 2.7. It can be computed that `AdaTest` engenders an average of 2.04× and 155.04× reduction of the test set size compared to MERO and TRIAGE across all benchmarks, respectively. The reduction of test set size has two benefits: (i) A smaller test set features a lower memory footprint; (ii) For on-chip test pattern generation, a smaller test set suggests a shorter test generation time.

Figure 2.13 compares the required test generation time of `AdaTest`, MERO, and TRIAGE to achieve the coverage results on various benchmarks in Table 2.7. Note that we use *log-scale* for the vertical axis since the range of runtime is diverse across different circuits. We can observe that `AdaTest` is the most efficient HT detection method among the three and it also achieves high Trojan coverage (last column of Table 2.7). More specifically, `AdaTest` engenders an average of 366.26× and 0.63× test generation speedup compared to MERO [61] and TRIAGE [62], respectively. Note that although the runtime of TRIAGE is smaller, its Trojan detection rate is 30% lower than `AdaTest`.

2.12.3 `AdaTest` Architecture Evaluation

The resource utilization of `AdaTest` depends on the input length and the circuit size. We report the resource utilization results of the evaluated benchmarks in Table 2.8. Figure 2.14 shows that `AdaTest` architecture achieves approximately linear speedup w.r.t. to the number of CEs. Our hardware design can be scaled up by adding more reward computing engines to parallel the circuit emulation process as `AdaTest`'s computation bottleneck is reward evaluation of the test patterns. Nevertheless, the speedup saturates when N_{CE} is sufficiently high. `AdaTest` broadcasts the wire values of the circuit response (given a test input) to all CEs via a shared data bus. Each CE scans the DAG buffer and obtains the broadcast wire values to compute the corresponding reward. Therefore, increasing the number of CEs does not lead to extra wire delay. However, more CEs suggest a higher overhead during reward accumulation.

Table 2.8. Resource utilization of the auxiliary circuitry on *c432*, *c880*, *c2670* and *des* benchmarks with default settings ($N_{CE} = 16$) on Zynq ZC706.

Benchmarks	c432	c880	c2670	des
BRAMS	26	36	65	237
DSP48E1	0	0	0	0
KLUTs (emulator usage)	14.9 (0.5)	25.5 (0.6)	61.1 (3.5)	267.9 (26.1)
FFs (emulator usage)	4,440 (80)	5,743 (160)	6,717 (317)	12,943 (1190)

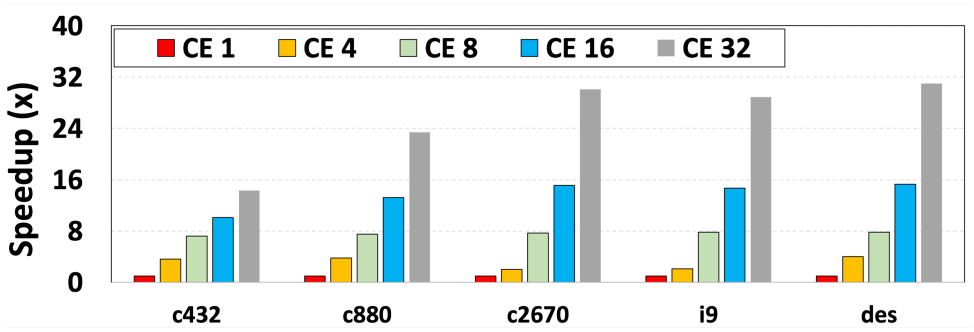


Figure 2.14. `AdaTest`'s scalability to the number of DAG reward computing engines. The speedup is near-linear with N_{CE} on large circuits where reward evaluation is the computation bottleneck.

2.13 Future Work

While groundbreaking, Trojan detection on neural networks presents several limitations that should be acknowledged. The first limitation is the runtime overhead. Even after using a GPU, the runtime still does not support frequent model evaluations, and future work is needed to make it even faster. One idea is to design a local detector within the current framework that responds more quickly and protects against known threats. However, there is a potential risk that a new attack method could fool the system and reduce detection accuracy.

For Hardware Trojans, a future research direction is to investigate the performance of AdaTest on other hardware security problems, such as logic verification and built-in self-test.

2.14 Conclusion

This chapter presents `DeepTD`, a Trojan detector for malicious DNN models implemented on an FPGA, which surpasses CPU and GPU designs in efficiency while utilizing fewer parameters. Our approach employs a scalable detector module for faster detection and lower power consumption. Our FPGA implementation achieves up to 60x faster detection throughput compared to CPU and GPU implementations and consumes 17x less power than the GPU implementation. Moreover, our method enables testing for safety assurance by running a self-test process in parallel with the model's inference. Our work also includes the development of reconfigurable detector modules that can accelerate various DNN architectures on hardware.

we also present a holistic solution to Hardware Trojan detection using adaptive, reinforcement learning-based test pattern generation. To formulate logic testing-based HT detection as an RL problem, we design an innovative reward function to characterize the quality of a test pattern from both static and dynamic aspects. `AdaTest` progressively expands the test set by identifying test input vectors with high reward values in an iterative approach.

`AdaTest` integrates adaptive sampling to identify and encourage high-reward test patterns, thus accelerating our RL-based input space exploration.

We devise `AdaTest` using a Software/Hardware co-design approach. Particularly, we develop a domain-specific system-on-chip architecture for efficient hardware implementation of `AdaTest`. Our architecture optimizes reward evaluation via circuit emulation and pipelines the computation of `AdaTest`. We perform extensive evaluations of `AdaTest` on various benchmarks and compare its performance with two counterparts, MERO and TRIAGE. Empirical results corroborate that `AdaTest` achieves superior effectiveness, efficiency, and scalability for HT detection compared to prior works. `AdaTest` is a *generic* test pattern generation framework, we plan to investigate its performance on other hardware security problems such as logic verification and built-in self-test in our future work.

2.15 Acknowledgements

This chapter is a partial reprint of the published material in H. Chen, X. Zhang, K. Huang, and F. Koushanfar, “AdaTest: Reinforcement learning and adaptive sampling for on-chip hardware Trojan detection,” in ACM Transactions on Embedded Computing Systems, vol. 22, no. 2, pp. 1-23, 2023. The dissertation author was the author of these papers.

Chapter 3

A scalable algorithm to improve the efficiency of Binary Neural Network

3.1 Introduction

There is an increasing surge in cloud-based inference services that employ deep learning models.

In this setting, the server trains and holds the DNN model, and clients query it to make inferences about their data. One major shortcoming of such a service is the leakage of clients' private data to the server, which can hinder commercialization in specific applications. For instance, in medical diagnosis [104], clients would need to expose their "plaintext" health information to the server, which violates patient privacy regulations such as HIPAA [105].

One attractive option for ensuring clients' content privacy is the use of modern cryptographic protocols, as they provide provable security guarantees [106, 107, 108, 109, 110, 111, 112, 113, 114, 2, 5]. Let $f(\theta, x)$ be the inference result on the client's input x using the server's parameters θ.

By executing cryptographically secure operations, the client and server can jointly compute $f(\theta, x)$ without revealing x to the server or θ to the client. We refer to this process as *oblivious inference* in the remainder of the paper. Unlike plaintext inference, oblivious inference protects the privacy of both parties. The challenge, however, is the excessive computation and communication overhead associated with privacy-preserving computation. For example, the contemporary state-of-the-art for performing oblivious inference on a single CIFAR-10 image requires an exchange of $\sim$ 3.4 GB of data and takes $\sim$ 10 seconds [1].

Early research on oblivious inference mainly focused on developing protocols for inference of a given DNN model without making significant modifications to the model itself [106, 107, 108, 109, 110, 111, 112, 113, 114, 2, 5]. Recently, a body of work has explored modifying the DNN architecture such that the resulting model is more amenable to secure computation [1, 4, 115, 3].

Other potential directions for enhancing oblivious inference could include pruning, tensor decomposition [116], quantization and Binary Neural Networks (BNNs) [117].

In this work, we study BNN as a candidate for fast and scalable oblivious inference. We show that a BNN has several unique characteristics that allow it to translate its computations into simple and efficient cryptographic protocols.

The benefits of employing BNNs for oblivious inference were first noted by XONN [1]. Despite achieving significant runtime improvement compared to non-binary DNN inference, XONN has not leveraged opportunities for translation.

Part of the inefficiency of XONN is due to the usage of a single secure computation protocol as a black box for all neural network layers after the input layer. In this work, we introduce a new hybrid approach in which the underlying secure computation protocol is customized to each layer, minimizing the total execution cost for oblivious inference on all layers.

We design a composite custom secure execution protocol optimized for BNN operations using standard security primitives. Our protocol significantly improves the efficiency of XONN, as shown in our experiments.

One standing challenge in oblivious BNN inference is finding network architectures that are accurate and amenable to secure computation. Since BNNs suffer from long training times and poor convergence, searching for such architectures could be inefficient.

We address the search inefficiency challenge by

Training a *single BNN* that can operate under different computational budgets. Our

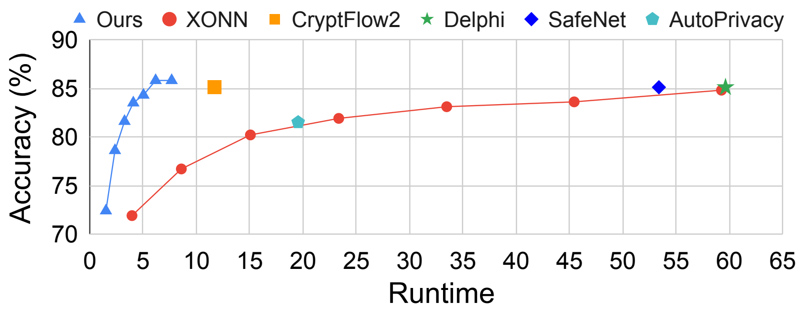


Figure 3.1. Accuracy and runtime of our oblivious BNN inference, compared with contemporary research with the same server-client scenario setting as ours (two-party, honest but curious). Among these, XONN [1] evaluates BNNs, whereas Cryptflow2 [2], Delphi [3], SafeNet [4], and AutoPrivacy [5] evaluate non-binary models.

adaptive BNN offers a tradeoff between accuracy and inference time without requiring training separate models. Figure 3.1 presents the tradeoff achieved by our flexible BNN on the 7-layer VGG network trained on CIFAR-10. With the combined power of our custom oblivious inference protocols and adaptive BNN training schemes, our method outperforms prior art regarding accuracy and runtime. For example, we achieve $\sim 2\times$ faster oblivious inference at the same accuracy compared to Cryptflow2 [2], the state-of-the-art non-binary DNN inference framework, and $2\times$ to $11\times$ lower runtime compared to XONN, the previous work on oblivious BNN inference.

We further improved the inefficiency challenge after combining neural architecture search and significantly improved our performance compared to our previous work [118].

The paper is organized as follows: Section 3.2 introduces the problem scenario and our threat model. Section 3.3 introduces the background of essential concepts. Section 3.4 presents the overview our proposed Cryptographically Secure BNN Inference method, specifically it contains linear layers, Nonlinear layers communication and discuss the security of our approach. Then, we discuss the communication cost. Next, Section 3.5 introduces our adaptive BNN training process to achieve excellent inference accuracy. Section 3.6 describes our experiment result with the state-of-the-art standard benchmarks and model architectures and the evaluation of private tasks. Section 3.7 summarizes the related work and discusses their drawbacks.

3.2 Scenario and Threat Model

Figure 3.2 presents the scenario in oblivious inference. Both the server and the client know the neural network architecture f.

The server holds the set of trained parameters, i.e., $\theta = \{\theta^1, \ldots, \theta^L\}$, and the client holds the input query to the neural network, i.e., x. The two parties use a secure function evaluation protocol, where the client learns the inference result $y = f(\theta, x)$.

Using a secure function evaluation protocol, the server and client jointly evaluate the neural network function $y = f(x_a, x_b)$ without revealing the server's weights to the client or the client's input to the server. At the end of the protocol,

Similar to prior work in privacy-preserving inference, we consider the honest-but-curious scenario [113, 114, 2, 5, 1, 4, 115, 3]. In this threat model, the two parties follow the protocol they agree upon to compute the output, yet they may try to learn as much as they can about the other party's data. As such, the protocol should guarantee the following security requirements:

- x or $f(\theta, x)$ are not revealed to the server.
- θ is not revealed to the client.
- Client and server do not learn intermediate activations.

Consistent with prior work, we assume that the model architecture is known to both parties.

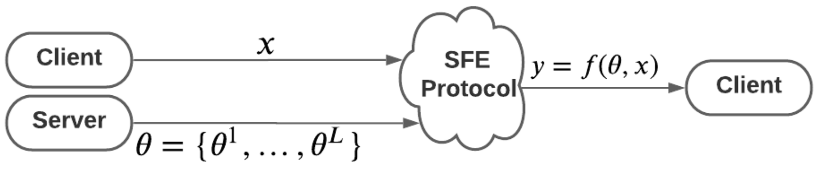


Figure 3.2. The server and client use a secure function evaluation (SFE) protocol to perform oblivious inference. At the end of the protocol, the client learns $y = f(\theta, x)$ without learning the server's parameters θ or revealing x to the server.

3.3 Background

This section provides a high-level outline of the necessary terminologies. Following the convention in secure computation literature, we refer to the server and client as Alice and Bob, respectively.

3.3.1 Secure Function Evaluation Protocol.

During oblivious inference, Alice and Bob engage in a Secure Function Evaluation (SFE) protocol, essentially a set of rules specifying their messages. By following these rules, they jointly compute the output of a function that takes the inputs from both without disclosing any information about Alice's data to Bob and vice versa. Depending on protocol agreements, the computation's result can be exposed to both parties, only one or neither.

Here is an example: Let's assume Alice and Bob have respective fortunes, A and B. They would like to play a game to determine who has more money, i.e., $A > B$, while at the same time, they do not want to reveal any additional information about their money. This prevalent task is the $millionaires'\ problem$[119]. In general, Alice and Bob, or more players, wish to compute the output of a function f on secret inputs without revealing any additional information from each player individually. In this case, f is called secure $function\ evaluation$ or $general\ secure\ multiparty\ computation.$

3.3.2 Additive Secret Sharing (AS)

Additive Secret Sharing (AS) is a method for distributing a secret x between Alice and Bob such that Alice holds $[\![x]\!]_A = x + r$ and Bob holds $[\![x]\!]_B = -r$, where r is a random value. Individually, both $[\![x]\!]_A$ and $[\![x]\!]_B$ are random values. Hence, Alice and Bob cannot decipher the original message x independently. Only by combining $[\![x]\!]_A$ and $[\![x]\!]_B$ can one recover the actual secret as $x = [\![x]\!]_A + [\![x]\!]_B$. Standard SFE protocols exist to perform addition and multiplication on secret-shared data, and the result is also shared between the two parties. We employ these protocols in oblivious inference to ensure that neither a layer's input nor output is revealed to the involved parties. We refer curious readers to [120] for more details.

3.3.3 Oblivious Transfer (OT)

Oblivious Transfer (OT) is a protocol between two parties – a sender who has two messages (μ_0, μ_1), and a receiver who has a selection bit $i \in \{0,1\}$. The receiver obtains the intended message μ_i through OT without revealing the selection bit i to the sender. The receiver does not learn the other message,μ_{1-i}. We refer curious readers to [121, 122, 123] for details about OT, its variants and their implementations.

In Section 3.4.1, we design a protocol for oblivious matrix multiplication, which enables oblivious evaluation of convolution and fully-connected layers. We build our protocol by only using oblivious transfer (OT) and additive secret sharing (AS), which we outlined above. However, AS and OT are inefficient for evaluating nonlinear activations and Max-pooling.

3.3.4 Garbled Circuit (GC)

Garbled Circuit (GC) is an SFE protocol that can evaluate an arbitrary function (linear or nonlinear). Unlike secret sharing, GC can evaluate *arbitrary* nonlinear functions. However, the downside of GC is its heavy communication overhead. Thus, we tend to limit its usage to nonlinear operations. The idea here is to describe the function f as a Boolean circuit, encrypt the nodes of this circuit, and perform. The input to the GC protocol is $[\![x]\!]_A$ by Alice and $[\![x]\!]_B$ by Bob. After GC evaluation, Alice and Bob receive $[\![y]\!]_A$ and $[\![y]\!]_B$, respectively. In other words, Alice and Bob jointly compute $y = f(x)$ without learning x or y. For instance, to securely add 32-bit scalars, Alice and Bob should exchange 8192 bits during the GC.

We refer curious readers to [124, 125, 126] for more details about GC. However, the downside of GC is its heavy communication overhead.

3.4 Cryptographically Secure BNN Inference

BNNs were originally introduced to minimize the memory footprint and computation overhead of plaintext inference. In this section, we provide insights into why BNNs are also helpful for very efficient and fast oblivious inference.

The first favorable property of BNNs is enforcing the weights to +1 or -1. With this restriction, multiplying a feature x by a weight w is equivalent to computing either $+x$ or $-x$. This simple property becomes useful when computing vector dot products of the form $\sum_{i=1}^{N} w_i x_i$, which can be computed via N conditional additions/subtractions. We show in Section 3.4.1 that conditional summations can be computed using OT and AS, which are known to be very efficient and lightweight cryptographic tools.

In oblivious inference, nonlinear operations are evaluated through heavy cryptographic primitives such as GC, resulting in significant runtime and communication overheads. The large communication cost of GC is directly related to the bit-widths of GC inputs. The second advantage of BNNs is their 1-bit hidden layer feature representation, which significantly reduces the GC evaluation cost compared to non-binary features. In Section 3.4.2, we expand on low-bit nonlinear operations and their efficient GC evaluation.

We present the overall flow for oblivious BNN inference in Figure 3.3. The inputs and outputs of all layers are in AS format, e.g., server and client have $[\![Y^i]\!]_A$ and $[\![Y^i]\!]_B$ rather than Y^i. To evaluate linear layers (CONV or FC) obliviously, we propose a novel custom protocol for binary matrix multiplication that directly works on AS data. We merge batch normalization (BN), binary activation (BA), and max-pooling (MP) into a single nonlinear function $f(\cdot)$. To securely evaluate $f([\![Y^i]\!]_A, [\![Y^i]\!]_B)$, three consecutive steps should be taken:

1. Securely translating the input from AS to GC. This step prepares the data to be processed by GC.

2. Computing the nonlinear layer through GC protocol.

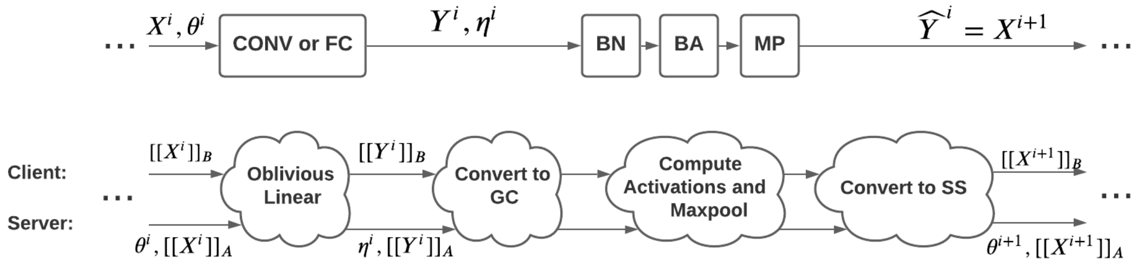


Figure 3.3. Illustration of plaintext inference (top) and our proposed equivalent oblivious inference (bottom). We denote linear layers by *CONV* and *FC*, Batch-Normalization by *BN*, Binary Activation by *BA*, and Max-Pooling by *MP*. Here, X^i, Y^i, and θ^i are the linear layer's input, output, and weight/bias parameters, respectively. η^i denotes BN parameters, and $\widehat{Y}$ is the output of binary activation. The client and the server perform the oblivious inference. To hide information, the input and output of a linear layer are in the AS domain, e.g., server and client have $[\![y^i]\!]_A$ and $[\![y^i]\!]_B$ rather than y^i. To evaluate nonlinear operations, the tuple $([\![y^i]\!]_A, [\![y^i]\!]_B)$ is first converted to GC ciphers of y^i. Then, GC is utilized to evaluate BN, BA, and MP. Next, the output of GC is converted back to AS-domain to serve as the input of the next layer.

3. Securely translating the result of the GC protocol to AS. This step prepares the data to be processed in the following linear layer.

We achieve a significantly faster oblivious inference using this hybrid approach compared to the state-of-the-art [1].

3.4.1 Linear Layers

Fully-connected and convolutional layers require computing $Y = WX$, with weight matrix W and input X. In secure matrix multiplication, the input is secretly shared between the server and the client, i.e., $X = [\![X]\!]_A + [\![X]\!]_B$. Bob (the client) has $[\![X]\!]_B$ whereas Alice (the server) has the weight W and $[\![X]\!]_A$[1]. The matrix multiplication is computed as follows:

$$W([\![X]\!]_A + [\![X]\!]_B) = W[\![X]\!]_A + W[\![X]\!]_B \tag{3.1}$$

Alice can compute $W[\![X]\!]_A$ locally, and only $W[\![X]\!]_B$ needs secure evaluation. After evaluating $Y = WX$,

- Alice gets $[\![Y]\!]_A$ but does not learn $[\![X]\!]_B$ or $[\![Y]\!]_B$.

[1] At the first layer, only the client has the input share, hence $[\![X]\!]_A = 0$

- Bob gets $[\![Y]\!]_B$ but does not learn W or $[\![Y]\!]_A$.

After obliviously evaluating $Y = [\![Y]\!]_A + [\![Y]\!]_B = WX$, Alice and Bob receive $[\![Y]\!]_A$ and $[\![Y]\!]_B$, respectively. For simplicity, we drop the subscripts hereafter and represent the matrices as $[\![W]\!]_A \rightarrow W$ and $[\![X]\!]_B \rightarrow X$.

Algorithm 5 presents the secure matrix multiplication protocol for the class of binary weights. Initially, Alice sets her output share to $W[\![X]\!]_A$ (line 7) and Bob sets his share to zero (line 8). Next, they obliviously evaluate $W[\![X]\!]_B$ one row at a time in the outer loop of Algorithm 5 (lines 9-21). Specifically, the m-th iteration of the outer loop evaluates the m-th row of the output as:

$$y = [\![y]\!]_A + [\![y]\!]_B = \sum_{n=1}^{N} W(m,n)X(n,:)$$

The inner loop of Algorithm 5 (lines 12-19) computes the above summation by N invocations of OT. After each OT invocation, Alice receives either $\mu_0 = r - [\![X(n,:)]\!]_B$ or $\mu_1 = r + [\![X(n,:)]\!]_B$ depending on the selection bit. It is easy to see that μ_i (known by Alice) and $-r$ (known by Bob) are the arithmetic shares of $W(m,n)[\![X(n,:)]\!]_B$.

Security. The security of our binary matrix multiplication is guaranteed as follows: first, OT guarantees that Alice's selection bit is not revealed to Bob. Hence, Alice's binary weight remains secret to her. Second, Bob adds a random vector to $[\![X(n,:)]\!]_B$ before participating in OT. After OT invocation, Alice receives one and only one of the two messages $\{r + [\![X(n,:)]\!]_B, r - [\![X(n,:)]\!]_B\}$, where r is random. Therefore, Alice cannot learn $[\![X(n,:)]\!]_B$ from the received message. Third, Alice does not communicate $[\![X]\!]_A$ to Bob. Thus, her input share is also kept private. Last, Alice and Bob do not communicate $[\![Y]\!]_A$ and $[\![Y]\!]_B$. Hence, their output shares are kept private.

3.4.2 Nonlinear Layers

In this section, we outline and leverage the characteristics of BNNs for oblivious inference of nonlinear layers. The cascade of batch normalization (BN) and binary activation (BA) takes input feature y and returns $\widehat{y} = sign(\alpha y + \beta) = sign(y + \frac{\beta}{\alpha})$, where α and β are the BN parameters.

Algorithm 5. Protocol for secure binary matrix multiplication

```
Input: from Alice $W \in \{-1,+1\}^{M\times N}$
Input: from Alice $[\![X]\!]_A \in \mathbb{Z}^{N\times L}$
Input: from Bob $[\![X]\!]_B \in \mathbb{Z}^{N\times L}$
Output: to Alice $[\![Y]\!]_A \in \mathbb{Z}^{M\times L}$
Output: to Bob $[\![Y]\!]_B \in \mathbb{Z}^{M\times L}$
Remark: $[\![Y]\!]_A + [\![Y]\!]_B = W([\![X]\!]_A + [\![X]\!]_B)$
Alice locally sets $[\![Y]\!]_A = W[\![X]\!]_A \in \mathbb{Z}^{M\times L}$
Bob locally sets $[\![Y]\!]_B = \mathbf{0} \in \mathbb{Z}^{M\times L}$
for $m \in [M]$ do
    Alice locally sets $[\![y]\!]_A = [\![Y]\!]_A(m,:)$
    Bob locally sets $[\![y]\!]_B = [\![Y]\!]_B(m,:)$
    for $n \in [N]$ do
        Bob generates random vector $r \in \mathbb{Z}^L$
        Alice and Bob engage in OT where:
            Bob inputs $\{\mu_0, \mu_1\} = \{r \pm [\![X]\!]_B(n,:)\}$
            Alice inputs $i = \frac{W(m,n)+1}{2}$
            Alice receives $\mu_i$
        Alice locally updates $[\![y]\!]_A = [\![y]\!]_A + \mu_i$
        Bob locally updates $[\![y]\!]_B = [\![y]\!]_B - r$
    Alice locally updates $[\![Y]\!]_A(m,:) = [\![y]\!]_A$
    Bob locally updates $[\![Y]\!]_B(m,:) = [\![y]\!]_B$
```

Since both α and β belong to the server, the parameter $\eta = \frac{\beta}{\alpha}$ can be computed offline. The GC evaluation of BN and BA only entails adding η to y and computing the sign of the result, which can be evaluated by relatively low GC cost [127].

Moreover, binary Max-Pooling can be efficiently evaluated at the bit level. Taking the maximum in a window of binarized scalars is equivalent to performing logical `OR` among the values, which is also efficient in GC [127].

Algorithm 6 presents our efficient protocol for oblivious evaluation of nonlinear layers in BNNs, which leverages the insights discussed above. Our protocol receives secret-shared data $[\![Y]\!]_A$, and batch-normalization parameter values $\eta = \frac{\beta}{\alpha}$ from the server, as well as $[\![Y]\!]_B$ from the client. It then computes $\widehat{Y}$ by applying batch normalization, binary activation, and max-pooling on Y. Upon completion of the protocol, the server and client receive $[\![\widehat{Y}]\!]_A$ and $[\![\widehat{Y}]\!]_B$, respectively, which they use to evaluate the proceeding layer.

Security. GC inherently guarantees the security of our protocol for inference of nonlinear layers.

During GC, no information about one party's input is revealed to the other party and vice versa, hence, $[\![Y]\!]_A$, $[\![Y]\!]_B$ and η are kept private to their owners. After GC, Alice receives $R+\widehat{Y}$. Alice does not know R, so she cannot recover $\widehat{Y}$. Bob knows the random share R but does not know $R+\widehat{Y}$ because GC returns it only to Alice. Therefore, Bob does not know the output either. Therefore, Bob's share $[\![\widehat{Y}]\!]_B = -R$ along with Alice's share $[\![\widehat{Y}]\!]_A = R+\widehat{Y}$ are secure additive shares.

Algorithm 6. Protocol for secure non-linear operations.

```
1:  Input: from Alice $[\![Y]\!]_A$
2:  Input: from Alice $\eta$
3:  Input: from Bob $[\![Y]\!]_B$
4:  Output: to Alice $[\![\widehat{Y}]\!]_A$
5:  Output: to Bob $[\![\widehat{Y}]\!]_B$
6:  Remark: $[\![\widehat{Y}]\!]_A + [\![\widehat{Y}]\!]_B = f([\![Y]\!]_A + [\![Y]\!]_B + \eta)$
7:  Remark: $f(\cdot)$ denotes BN, BA, and optional MP.
8:  Alice locally computes $[\![Y]\!]_A + \eta$
9:  Bob locally generates random tensor $R$
10: Alice and Bob engage in GC where:
11:     Alice inputs $[\![Y]\!]_A + \eta$
12:     Bob inputs $[\![Y]\!]_B$ and $R$
13:     GC computes $F = R + f([\![Y]\!]_A + \eta + [\![Y]\!]_B)$
14:     GC returns $F$ only to Alice
15: Alice sets $[\![\widehat{Y}]\!]_A = F$
16: Bob sets $[\![\widehat{Y}]\!]_B = -R$
```

3.4.3 Communication Cost

Recall that each layer execution is done via SFE protocol, where the two involved parties cooperatively compute output shares of their own. During the protocol, each party may perform certain computations, storage, or random data generation internally on their device. These local processes are deemed as *free* operations in privacy-preserving computation. In practice, the process's runtime is dominated by the exchange of messages between the two parties, not the internal computations. In our protocols (Algorithms 5& 6), message exchanges occur during OT or GC invocations. e provide the coprocess's runtime protocols in Table 3.1. By plugging in the

Table 3.1. Communication Cost for different stages of our oblivious inference protocols. For matrix multiplication, N, M, L are the dimensions from $W_{M\times N}$ and $X_{N\times L}$, and b is the bitwidth for arithmetic sharing[2]. For batch normalization and binary activation, κ is a security parameter, and its standard value is 128 in the literature. For max-pooling, w is the window size, and $L' \approx \frac{L}{w^2}$ is represented with a different notation than L to account for the lower output resolution. Depending on whether or not max-pooling was applied, the secret sharing cost can be $3\kappa bML'$ or $3\kappa bML'$.

Stage	Underlying Operation	Communication (bits)
Mat-Mult	$[\![Y]\!]_A + [\![Y]\!]_B = W([\![X]\!]_A + [\![X]\!]_B)$	$NbML$
BN+BA	$\widehat{Y} = sign([\![Y]\!]_A + \eta + [\![Y]\!]_B)$	$5\kappa bML$
Max Pooling	$\widehat{Y} \leftarrow maxpool_{w\times w}(\widehat{Y})$	$2(w^2-1)\kappa ML'$
Secret Sharing	$[\![\widehat{Y}]\!]_A \leftarrow \widehat{Y} + R$	$3\kappa bML'$ or $3\kappa bML$

parameters of this table, one can compute the total execution cost for oblivious inference of a given BNN architecture. As we show in our experiments, the communication cost is closely tied to the runtime of our protocols.

3.5 Training Adaptive BNN

One of the primary challenges of BNNs is to ensure inference accuracy comparable to the non-binarized model. Since the introduction of BNNs, there have been tremendous efforts to improve inference accuracy by increasing the number of channels per convolution layer [128], increasing the number of computation bits [129], or introducing new connections and nonlinear layers [130, 131], to name a few. In this paper, we improve the accuracy of the base BNN by multiplying its width, e.g., by training an architecture with twice as many neurons at each layer. In practice, specifying the appropriate width for a BNN architecture requires exploring models with various widths, which can be quite time-consuming and cumbersome. Each model with a certain width should be trained and stored separately. What aggravates the problem is that BNNs suffer from convergence issues unless the data augmentation and training hyperparameters are carefully selected [132].

[2]To ensure correctness, b should be set to $\lceil Log(N)+1 \rceil$. In practice, software libraries only support multipliers of 8. Hence, we set b to the smallest multiplier of 8 bigger than or equal to $\lceil Log(N)+1 \rceil$.

A related field of research is training dynamic DNNs [133], with the goal of providing flexibility at inference time. In this realm, we find Slimmable Networks [134] quite compatible to our problem setting and adapt them to BNNs. Our goal is to train a single network with certain maximum width, say $4\times$ the base network, in a way that the model can still deliver acceptable accuracy at lower widths, e.g., $1\times$ or $2\times$ the base network. Once this model is trained, it can operate under any of the selected widths, thus, providing a tradeoff between accuracy and runtime.

Slimmable BNNs Definition. Let us denote the base BNN as M_1 and represent BNNs with $s\times$ higher width at each layer with M_s. Our goal is to train $M_{s_1} \subset M_{s_2} \subset M_{s_n}$ for a number of widths $\{s_i\}_{i=1}^{n}$. The weights of M_{s_i} are a subset of the weights of $M_{s_{i+1}}$. Therefore, having M_{s_n} we can configure it to operate as any M_{s_i} for $i \leq n$.

Training Slimmable BNNs. For a given minibatch X, each subset model computes the output as $\widetilde{Y}_{s_i} = M_{s_i}(X)$, resulting in $\{\widetilde{Y}_{s_1}, \ldots, \widetilde{Y}_{s_n}\}$ computed by $M_{s_1} \ldots M_{s_n}$. The ground-truth label Y is then used to compute the cumulative loss function as $\sum_{i=1}^{n} \mathscr{L}(Y, \widetilde{Y}_{s_i})$, where $\mathscr{L}(\cdot,\cdot)$ represents cross-entropy. The BNN weights are then updated using the standard gradient approximation rule suggested in [117].

3.6 Evaluations

Standard Benchmarks. We perform our evaluation on several networks trained on the CIFAR-10 dataset, shown in Table 3.2. These benchmarks provide us with a rich set of comparison baselines as they are commonly used in prior work. Specifically, the BC1 network has been evaluated by the majority oblivious inference papers [113, 135, 136, 114, 1, 3, 2, 4, 5]. Other models are evaluated by XONN [1], the state-of-the-art for oblivious inference of *binary* networks. For brevity, we omit details about layer-wise configurations and refer curious readers to [1] for further information.

Table 3.2. Summary of the trained binary network architectures evaluated on the CIFAR-10 dataset.

Arch.	**Previous Papers**	**Description**
BC1	[113], [135], [136], [114], [1], [3], [2], [4], [5]	7 CONV, 2 MP, 1 FC
BC2	[1]	9 CONV, 3 MP, 1 FC
BC3	[1]	9 CONV, 3 MP, 1 FC
BC4	[1]	11 CONV, 3 MP, 1 FC

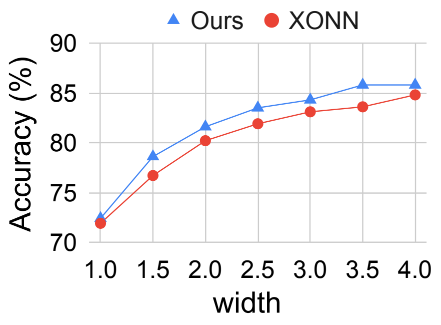
Training. For all benchmarks, we use standard backpropagation algorithm proposed by [117] to train our binary networks. We split the CIFAR10 dataset to 45k training examples, 5k validation examples, and 10k testing examples, and train each architecture for 300 epochs. We use Adam optimizer with initial learning rate of 0.001, and the learning rate is multiplied by 0.1 after 101, 142, 184 and 220 epochs. The batch size is set to 128 across all CIFAR10 training experiments. The training data is augmented by zero padding the images to 40×40, and randomly cropping a 32×32 window from each zero-padded image.

Evaluation Setup. The training codes are implemented in Python using the Pytorch Library. We use a single Nvidia Titan Xp GPU to train all benchmarks. We design a library for oblivious inference in C++. For implementation of OT and GC, we use the standard emp-toolkit [137] library. To run oblivious inference, we translate the model description and trained parameters from Pytorch to the equivalent description in our C++ library. For measurements, we run our oblivious inference code on a computer with 2.2 GHz Intel Xeon CPU and 16 GB RAM.

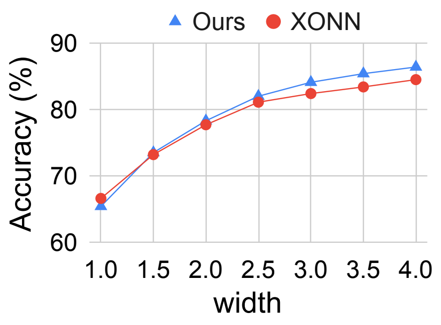
3.6.1 Evaluating Flexible BNNs

Let us start by evaluating our adaptive BNN training. We train slimmable networks with maximum $4\times$ width of the base models presented in Table 3.2. During training, we re-iterate through subsets of widths $\{1\times, 1.5\times, \ldots, 4\times\}$ and perform gradient updates as explained in Section 3.5.

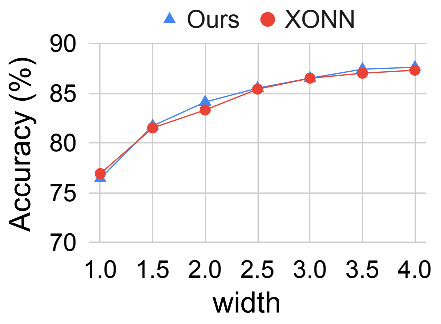
Figure 3.4 presents the test accuracy of each network at different widths. We also report the accuracy of independetly trained networks reported by XONN. The test accuracy



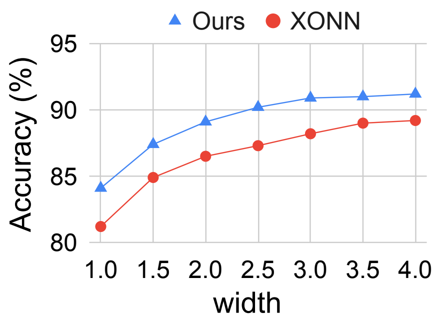
(a) BC1



(b) BC2



(c) BC3



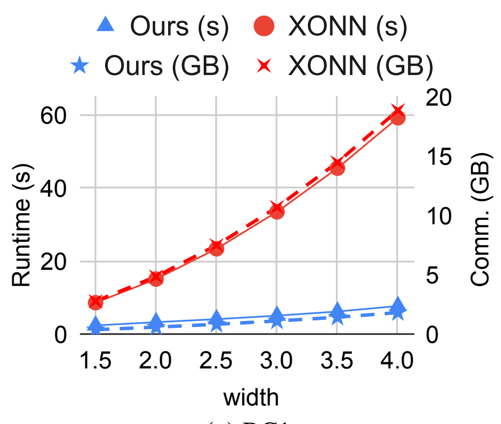
(d) BC4

Figure 3.4. CIFAR-10 test accuracy of each architecture at different widths. Our Adaptive BNN trains a single network that can operate at all widths, whereas previous work (XONN) trains a separate BNN per width.

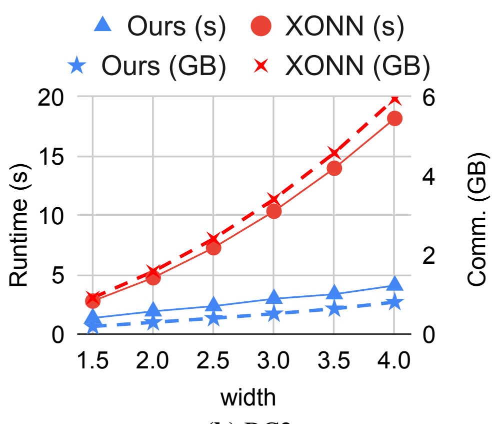
of a particular base BNN architecture can be improved by increasing its width. Our adaptive networks obtain better accuracy than independently trained BNNs at each width. Once the adaptive network is trained, the server can provide oblivious inference service to clients, which we discuss in the following section.

3.6.2 Oblivious Inference

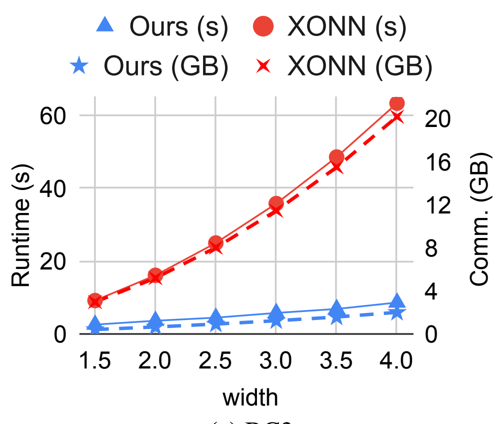
Recall that the runtime of oblivious inference is dominated by data exchange between client and server. We compare the communication cost and runtime of our custom protocol with XONN's GC implementation in Figure 3.5. The horizontal axis in each figure presents the network width. The left and right vertical axes respectively show the runtime (in seconds) and communication (in Giga-Bytes). The figure shows that for all the benchmarks, the runtime and communication of our method are significantly smaller than XONN. As seen, increasing the network width results in higher communication and runtime, which is the cost we pay for higher



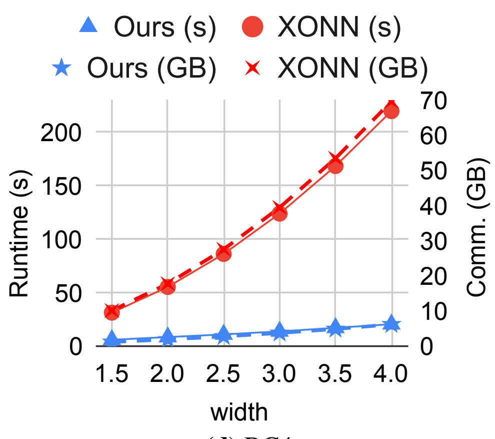
(a) BC1



(b) BC2



(c) BC3

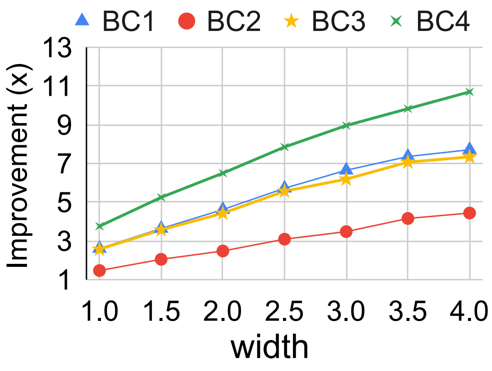


(d) BC4

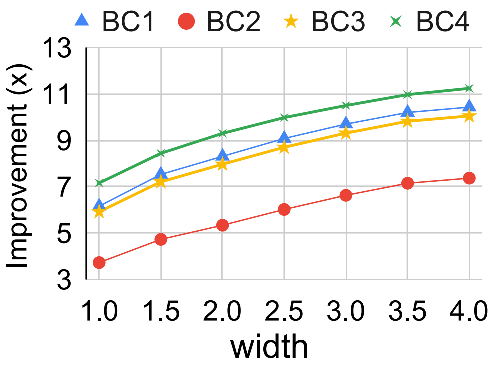
Figure 3.5. Runtime and communication cost of each architecture at different widths.
inference accuracy.

Figure 3.6 summarizes the performance boost achieved by our protocols, i.e., $2\times$ to $11\times$ lower runtime and $4\times$ to $11\times$ lower communication compared to XONN. The enhancement is more significant at higher widths, which shows the scalability for our method. To illustrate the reason behind our protocol's better performance, we focus our attention to the BC2 network at width 2.5, and show the breakdown of its communication cost in Figure 3.7. For the XONN protocol, most of the cost is from linear operations, which we reduce from 2.16GB to 0.15GB. In nonlinear layers, our cost is slightly more that XONN's, i.e., 0.25GB versus 0.09GB, which is due to the extra cost of conversion between AS and GC. Overall, the total communication is reduced from 2.25GB to 0.4GB compared to XONN.

Comparison to Non-binary Models. Among the architectures presented in Table 3.2, BC1



(a) Runtime



(b) Communication

Figure 3.6. Improvements in LAN runtime and communication compared to XONN. Our protocols achieve $2\times$ to $11\times$ in runtime and $4\times$ to $11\times$ communication reduction.

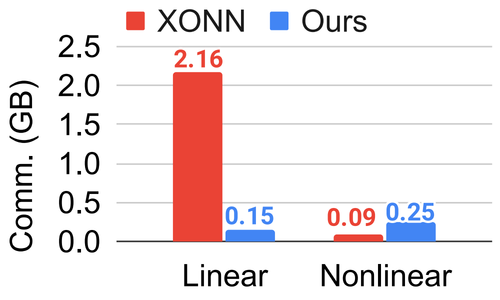


Figure 3.7. Breakdown of communication cost at linear and nonlinear layers for BC2 network. Our protocol significantly reduces XONN's GC-based linear layer cost, with a slight increase in nonlinear layer cost.

has been commonly evaluated in contemporary oblivious inference research. In Figure 3.1 we compare the performance of our method to the best-performing earlier work on this benchmark. The vertical and horizontal axes in the figure represent test accuracy and runtime, hence, points to the top-left corner are more desirable. Our method achieves a better accuracy/runtime tradeoff than all contemporary work while providing flexibility. Compared to Cryptflow2 (the most recent oblivious inference framework at the time of this paper), our method achieves $\sim 2\times$ faster inference at the same accuracy.

Evaluation in Wide Area Network (WAN). So far we reported our runtimes for the setting where client and server are connected via LAN, which is the most common assumption among prior work. We now extend our evaluation to the WAN setting, where the bandwidth is $\sim$ 20MBps and the delay is $\sim$ 50ms. The aforesaid bandwidth and delay correspond to the connection speed

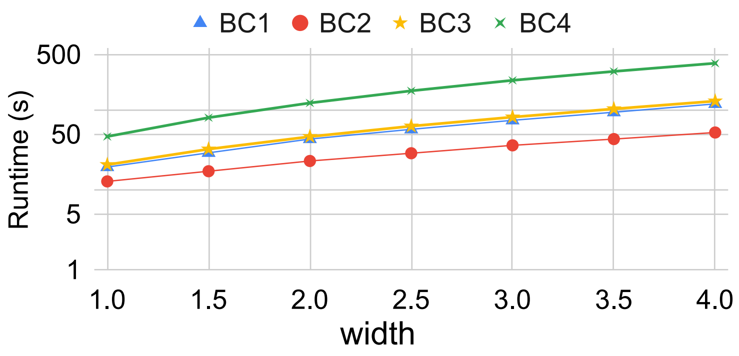


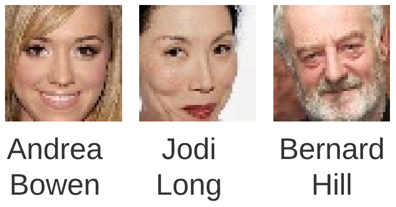
Figure 3.8. Runtime in WAN setting with ~ 20 MBps bandwidth and ~ 50 ms network delay.

between two AWS instances located in "US-West-LA-1a" and "US-East-2a". Runtimes are reported in Figure 3.8, showing varying inference time from 13 to 367 seconds depending on architecture and width. The results show the great potential of BNNs for commercial use. Indeed, the delay introduced by oblivious inference might not be tolerable in many applications that require real-time response, e.g., Amazon Alexa. However, there exist many applications where guaranteeing privacy is much more crucial than runtime, and several seconds or even minutes of delay can be tolerated. We evaluate two such applications in the following section.

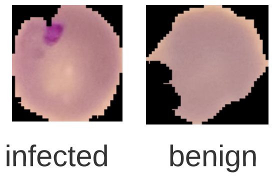
3.6.3 Evaluation on Private Tasks

In this section, we study the application of oblivious inference in face authentication and medical data analysis. Both applications involve sensitive features that the client wishes to keep secret: revealing medical data is against the HIPPA [105] regulation, and facial features can be used by malicious hackers to authenticate into the client's personal accounts. Since we do not have access to real private data, our best choice is to simulate these tasks using similar datasets that are publicly available to the research community. We evaluate our method on FaceScrub [138, 139] and Malaria Cell Infection [140] as representatives for face authentication and medical diagnosis, respectively.

Figure 3.9 shows example samples from each dataset. The tasks are to identify 530 different actors in Facescrub and classify infected cells from benign ones in Malaria. We were able to download $\sim 57,000$ images from the links provided by FaceScrub authors, of which we use 45000 for training, 6000 for validation, and 6000 for testing. The Malaria dataset is split



(a) Facescrub



(a) Malaria Cells

Figure 3.9. examples of input samples and labels from each dataset. For training, we resize Facescrub and Malaria cell images to 50×50 and 32×32, respectively.

to $\sim$ 24800 samples for training, $\sim$ 1300 for evaluation, and $\sim$ 1300 for testing. We train the *BC2* architecture at width 3 and 1 on FaceScrub and Malaria. The accuracy and performance results in the WAN setting are summarized in Table 3.3. Our model reaches 53.1% inference accuracy on FaceScrub and 92.4% accuracy on Malaria infection detection. The networks incur runtimes of 1-3 and 10-30 seconds in LAN and WAN settings, showing great potential for practical deployment. Note that in a commercial application the network architecture can be selected more carefully and more training data can be collected to achieve a better accuracy and runtime.

Table 3.3. Example BNNs trained for face recognition and medical application. We use the *BC2* architecture at width 3 and 1 for FaceScrub and Malaria, respectively. Runtimes are measured in the WAN setting.

Task	**Classes**	**Accuracy**	**Comm.**	**Runetime (s)**	
				LAN	**WAN**
FaceScrub	530	70.8%	404 MBs	2.2	32.2
Malaria	2	94.7%	80.5 MBs	0.7	11.5

3.7 Related Work

Oblivious inference was shown to be conceptually practical for small sized neural networks in CryptoNets [141]. Using CryptoNets, an inference on MNIST data would take $\sim$ 300 seconds, which motivated researcher to invest in the field. Since then, a plethora of more efficient protocols for oblivious inference have been proposed [106, 107, 108, 109, 110, 111, 112, 113, 114, 2, 5]. These works mainly focus on optimization of security primitives for oblivious inference, without making major modifications to the model.

A second line of research has been focused on identifying DNN models that are inherently amenable to secure execution protocols. Several DNN modification examples include replacing ReLU operations with square function [141, 3, 4], using dimensionality reduction at the input layer [142], and neural architecture search [115]. Concurrently, researchers in ML community have devised DNN optimization techniques such as pruning [143], quantization [144], tensor factorization [116], and binary neural networks [117]. Among the above, BNNs are especially compelling candidates for oblivious inference, since they translate linear arithmetic to bitwise operations. XONN [1] was the first work to notice the especial use case of binary networks for cryptographically secure inference using GC [125], noting that `XNOR` operations that frequently appear in BNNs can be evaluated for free in GC.

Despite improving oblivious inference time, XONN does not completely utilize the full set of opportunities provided by BNNs. Instead of using GC as a black box, we propose a hybrid protocol where GC is only used for nonlinear operations. We propose a novel protocol for matrix multiplication based on secret sharing and oblivious transfer. By exploiting the characteristics of BNN linear operations, our protocol achieves up to $11\times$ reduction in runtime compared to XONN. A remaining challenge with BNNs is their low inference accuracy, which XONN addresses partially by brute-force training of many BNN models, and choosing the one with proper accuracy/runtime for deployment. Alternatively, we show that BNNs can be trained via the Slimmable Network training technique [134]. We provide accurate and efficient BNN benchmarks for oblivious inference, that offer a tradeoff between execution cost and inference accuracy.

3.8 Future Work

Variants of BNNs are being developed to enhance inference accuracy, opening exciting avenues for future research. Developing custom protocols to securely evaluate residual connections [130], residual activation binarization [129], and PReLU nonlinearity [131] are interesting future directions for oblivious BNN inference. Our current oblivious inference implementation does not support these operations. However, since we use GC for non-linear operations and GC can implement arbitrary functionalities, the aforementioned techniques can be integrated and tested in future work, which may or may not result in improved accuracy-runtime tradeoff.

3.9 Conclusion

This chapter studies the application of binary neural networks in oblivious inference, where a server provides a privacy-preserving inference service to clients. Using this service, clients can run the neural network owned by the server, without revealing their data to the server or learning the parameters of the model. We explore favorable characteristics of BNNs that make them amenable to oblivious inference, and design custom cryptographic protocols to leverage these characteristics. In contrast to XONN [1], which uses GC to evaluate both linear and non-linear layers, we use GC only for nonlinear layers. We present a custom protocol for linear layers using OT and AS, which leads to $2\times$ to $11\times$ performance improvement compared to XONN. We also address the problem of low inference accuracy by training adaptive BNNs, where a single model is trained to be evaluated under different computational budgets. Finally, we extend our evaluations to computer vision tasks that perform inference on private data, i.e., face authentication and medical data analysis.

3.10 Acknowledgements

This chapter is, in part, a reprint of the published material in M. Samragh, S. Hussain, X. Zhang, K. Huang, and F. Koushanfar, “On the application of binary neural networks in oblivious inference,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4630-4639, 2021. and X. Zhang, M. Samragh, S. Hussain, K. Huang, and F. Koushanfar, “Scalable Binary Neural Network applications in Oblivious Inference,” in ACM Transactions on Embedded Computing Systems, vol. 23, no. 3, pp. 1-18, 2024, The dissertation author was the (co)primary investigator and author of these papers.

Chapter 4

Advancing Robustness in Federated Learning Environments

4.1 Introduction

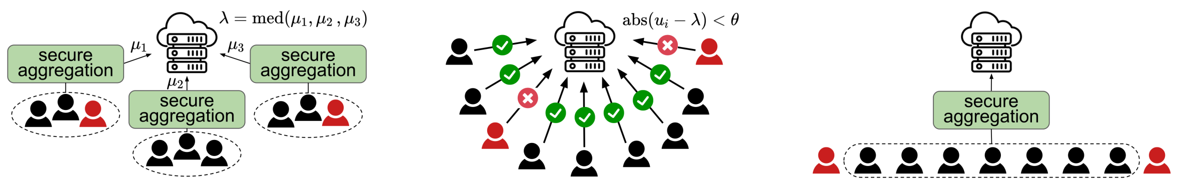
Federated learning (FL) has emerged as a popular paradigm for training a central model on a dataset distributed amongst many parties, by sending model updates and without requiring the parties to share their data. However, model updates in FL can be exploited by adversaries to infer properties of the users' private training data [17]. This lack of privacy prohibits the use of FL in many machine learning applications that involve sensitive data such as healthcare information [18] or financial transactions [19]. As such, existing FL schemes are augmented with privacy-preserving guarantees. Recent work propose secure aggregation protocols using cryptography [6]. In these protocols, the server does not learn individual user updates, but only a final aggregate with contribution from several users. Hiding individual updates from the server opens a large attack surface for malicious clients to send invalid updates that compromise the integrity of distributed training.

Some commonly known attacks on FL include Byzantine attacks, attribute inference attacks and poisoning attacks. They represent significant threats to machine learning systems, especially in distributed and decentralized environments. *Byzantine attacks* on FL are carried out by malicious clients who manipulate their local updates to degrade the model performance [145, 146, 147, 148]. Popular high-fidelity Byzantine-robust aggregation rules rely on rank-based statistics, e.g., trimmed mean [149], median [149], mean around median [150], and geometric median [151]. These schemes require sorting of the individual model updates across users. As such, using them in secure FL is nontrivial and unscalable to large number of users since the central server cannot access the (plaintext) value of user updates.

Attribute inference attacks in FL are privacy threats where an adversary attempts to infer sensitive attributes or characteristics of participants' data by analyzing the model updates exchanged during the collaborative learning process. These attacks exploit unintended information leakage from the model updates to violate users' privacy. For example, by analyzing the gradients or weight updates shared during Federated Learning, an adversary may identify patterns or correlations between certain gradient values and specific attributes. This information can then be used to infer the presence or absence of those attributes in the participants' local datasets. To mitigate attribute inference attacks in FL, various privacy-preserving techniques can be employed. Differential Privacy, for instance, is a widely used approach that adds noise or randomness to the model updates before aggregation, ensuring that the information leakage from individual updates is minimized. Secure Multi-Party Computation (MPC) and Homomorphic Encryption are other techniques that can be used to perform secure aggregation or protect the model updates' privacy during the aggregation process[152].

Poisoning attacks in collaborative learning involve adversaries injecting malicious data into participants' training sets, compromising the model's accuracy and integrity, and potentially biasing it towards the adversary's goals. However, they are just one of many threats in distributed systems. Focusing only on poisoning attacks leaves systems vulnerable to others, like Byzantine attacks, which could be more damaging. Comprehensive security requires addressing a range of attack vectors [153].

In this work we address aforementioned challenges and provide high break point Byzantine tolerance using rank-based statistics while preserving privacy. We propose a median-based robustness check that derives a threshold for acceptable model updates using securely computed mean over random user clusters. Our thresholds are dynamic and automatically change based



(a) Step 1: The server randomly clusters the users, obtains cluster means μ_i, and computes the median of cluster means (μ_is).

(b) Step 2: Each client provides a ZKP attesting that their update is within the threshold from the median of cluster means.

(c) Step 3: Clients marked as benign participate in a final round of secure aggregation and the server obtains the result.

Figure 4.1. High level description of `zPROBE` robust and private aggregation.

on the distribution of the gradients. Notably, we do not need access to individual user updates or public datasets to establish our defense. We leverage the computed thresholds to identify and filter malicious users in a privacy-preserving manner. Our Byzantine-robust framework, `zPROBE`, incorporates carefully crafted zero-knowledge proofs [154] to check user behavior and identify possible malicious actions, including sending Byzantine updates or deviating from the secure aggregation protocol. As such, `zPROBE` guarantees correct and consistent behavior in the challenging malicious threat model.

`zPROBE` integrates probabilistic optimizations for efficient zero-knowledge checks without sacrificing security. This unique co-design of robustness defense and cryptography enables a scalable, low-overhead approach for private, robust Federated Learning (FL). It's the first to have costs growing sub-linearly with client numbers. `zPROBE` achieves sub-second client compute time in aggregation rounds on ResNet20 over CIFAR-10. We demonstrate `zPROBE`'s performance on three foundational computer vision benchmarks, underscoring its scalability to larger benchmarks. In summary, our contributions are:

- Developing a novel privacy-preserving robustness check based on rank-based statistics. `zPROBE` is robust against various Byzantine attacks with $0.5-2.8\%$ higher accuracy compared to prior works.
- Enabling private and robust aggregation in the malicious threat model by incorporating zero-knowledge proofs. Our Byzantine-robust secure aggregation, for the first time, scales sub-linearly with client number.

- Leveraging probabilistic optimizations to reduce `zPROBE` overhead without compromising security, resulting in orders of magnitude client runtime reduction.

4.2 Cryptographic Primitives

Shamir Secret Sharing [155] is a method to distribute a secret s between n parties such that any t shares can be used to reconstruct s, but any set of $t-1$ or fewer shares reveal no information about the secret. Shamir's scheme picks a random $(t-1)$-degree polynomial P such that $P(0)=s$. The shares are then created as $(i, P(i)), i \in \{1, \ldots, n\}$. With t shares, Lagrange Interpolation can be used to reconstruct the polynomial and obtain the secret.

Zero-Knowledge Proof (ZKP) is a cryptographic primitive between two parties, a prover $\mathscr{P}$ and a verifier $\mathscr{V}$, which allows $\mathscr{P}$ to convince $\mathscr{V}$ that a computation on $\mathscr{P}$'s private inputs is correct without revealing the inputs. We use the Wolverine protocol [154] with highly efficient $\mathscr{P}$ in terms of runtime, memory usage, and communication.

In Wolverine, value x known by $\mathscr{P}$ can be authenticated using information-theoretic message authentication codes (IT-MACs) [156] as follows: assume Δ is a global key sampled uniformly and is known only to $\mathscr{V}$. $\mathscr{V}$ is given a uniform key $K[x]$ and $\mathscr{P}$ is given the corresponding MAC tag $M[x]=K[x]+\Delta.x$. An authenticated value can be *opened* (verified) by $\mathscr{P}$ sending x and $M[x]$ to $\mathscr{V}$ to check whether $M[x] \overset{?}{=} K[x]+\Delta.x$. Wolverine represents the computation as either an arithmetic or Boolean circuit, where the secret wire values are authenticated as described. The circuit is evaluated jointly by $\mathscr{P}$ and $\mathscr{V}$, culminating in $\mathscr{P}$ opening the output to indicate proof correctness.

Secure FL Aggregation includes a server and n clients each holding a private vector of model updates with l parameters $\boldsymbol{u} \in \mathbb{R}^l$. The server wishes to obtain the aggregate $\sum_{i=1}^{n} \boldsymbol{u_i}$ without learning any of the individual client updates. [6] proposes a secure aggregation protocol using low-overhead cryptographic primitives such as one-time pads. Each pair of clients (i, j) agrees on a random vector $\boldsymbol{m}_{i,j}$. User i adds $\boldsymbol{m}_{i,j}$ to their input, and user j subtracts it from their input so the masks cancel out when aggregated. To ensure privacy in case of dropout or network delays, each user adds an additional random mask $\boldsymbol{r_i}$. Users then create t-out-of-n Shamir shares of their masks and share them with other clients. User i computes their masked input as follows:

$$\boldsymbol{v_i} = \boldsymbol{u_i} + \boldsymbol{r_i} - \sum_{0<j<i} \boldsymbol{m}_{i,j} + \sum_{i<j\leq n} \boldsymbol{m}_{i,j} \tag{4.1}$$

Once the server receives all masked inputs, it asks for shares of pairwise masks for dropped users and shares of individual masks for surviving users (but never both) to reconstruct the aggregate value. The construction in [6] builds over [152] and improves the client runtime complexity to logarithmic scale rather than linear with respect to the number of clients. Note that the secure aggregation of [6] assumes the clients are semi-honest and do not deviate from the protocol. However, these assumptions are not suitable for our threat model which involves malicious clients. We propose an aggregation protocol that benefits from speedups in [6] and is augmented with zero-knowledge proofs for the challenging malicious setting as described below.

4.3 Methodology

Threat Model. We aim to protect the privacy of individual client updates as they leak information about clients' private training data. No party should learn any information about a client's update other than the contribution to an aggregate value with inputs from a large number of other clients. We also aim to protect the central model against Byzantine attacks, i.e., when a malicious client sends invalid updates to degrade the model performance. We consider a semi-honest server that follows the protocol but may try to learn more information from the received data. We assume a portion of clients are malicious, i.e., arbitrarily deviating from the protocol, or sending erroneous updates to cause divergence in the central model. Notably, we assume the clients may: ① perform Byzantine attacks by changing the value of their model update to degrade central model performance, ② use inconsistent update values in different steps of the secure aggregation protocol, ③ perform the masked update computation in Eq. 4.1 incorrectly or with wrong values, and ④ use incorrect seed values in generating masks and shares.

Algorithm 7. `zPROBE` secure aggregation

```
Input: Shamir threshold value $t$, clients set $U$
Round 1: Mask Generation
client i: Generate key pair $(sk_i, pk_i)$, sample $b_i$
        $a_{i,j} \leftarrow$ KeyAgreement$(sk_i, pk_j)$
        $\boldsymbol{m}_{i,j} \leftarrow$ PRG$(a_{i,j})$, $\boldsymbol{r_i} \leftarrow$ PRG$(b_i)$
        $\{s_j^{sk}\}_{j\in U}, \leftarrow$ SS$(sk_i, t)$, $\{s_j^b\}_{j\in U} \leftarrow$ SS$(b_i, t)$
        Send $s_j^{sk}, s_j^b$ to client $j$
Round 2: Update Masking
client i: $\boldsymbol{v_i} \leftarrow \boldsymbol{u_i} + \boldsymbol{r_i} - \sum_{0<j<i} \boldsymbol{m}_{i,j} + \sum_{i<j\leq U} \boldsymbol{m}_{i,j}$
        Authenticate $\boldsymbol{u_i}$ and send $\boldsymbol{v_i}$ to server
        Perform correctness check in Alg 8
server: Sample $q$ indices $S_i$ (Sec. 4.3.4) for client $i$
        Perform correctness check in Alg 8
Round 3: Aggregate Unmasking
server: $U_d \leftarrow$ dropped clients, $U_s \leftarrow$ surviving clients
        Collect $t$ shares of $\{s_i^{sk}\}_{i\in U_d}$ and $\{s_i^b\}_{i\in U_s}$
        Agg $\leftarrow \sum_{i\in U_s} \boldsymbol{v_i} - \sum_{i\in U_s} \boldsymbol{r_i} + \sum_{i\in U_s, j\in U_d} \boldsymbol{m}_{i,j}$
```

To the best of our knowledge, `zPROBE` is the first single-server framework with malicious clients that is resilient against such an extensive attack surface, supports client dropouts, and does not require a public clean dataset.

4.3.1 `zPROBE` Overview

`zPROBE` comprises two main components, namely, secure aggregation, and robustness establishment. We propose a new secure aggregation protocol for malicious clients in Sec. 4.3.2. Our proposed method to establish robustness is detailed in Sec. 4.3.3. We design an adaptive Byzantine defense that finds the dynamic range of acceptable model updates per iteration. Using the derived bounds, we perform a secure range check on client updates to filter Byzantine attackers. Our robustness check is privacy-preserving and highly scalable.

The proposed robust and private aggregation is performed in three steps as illustrated in Fig. 4.1. First the server clusters the clients randomly into c clusters. Each cluster c_j then performs `zPROBE`'s secure aggregation protocol. The server obtains the aggregate value $\boldsymbol{\alpha_j}$ and the mean $\boldsymbol{\mu_j} = \boldsymbol{\alpha_j}/|c_j|$ for each cluster in plaintext. In the second step, the server uses

Algorithm 8. Circuit for `zPROBE` correctness check

```
**Client input:** $b_i$, $a_{i,j}$, authenticated $\boldsymbol{u_i}$
**Public input:** $\boldsymbol{v_i}$, indices set $S_i$, clients set $U$
$check = 1$
**for** $k$ in $S_i$
    $\hat{r}_i^k \leftarrow \text{PRG}^k(b_i)$
    **for** $j$ in $U$
        $\hat{m}_{i,j}^k \leftarrow \text{PRG}^k(a_{i,j})$
    $\hat{v}_i^k \leftarrow u_i^k + \hat{r}_i^k - \sum_{0<j<i} \hat{m}_{i,j}^k + \sum_{i<j\leq n} \hat{m}_{i,j}^k$
    $check = check \wedge (\hat{v}_i^k = v_i^k)$
**return** $check$
```

Algorithm 9. Circuit for `zPROBE` robustness check

```
**Client input:** Authenticated $\boldsymbol{u_i}$
**Public input:** $\boldsymbol{\lambda}$, $\boldsymbol{\theta}$, indices set $S_i$
$check = 1$
**for** $k$ in $S_i$
    $check = check \wedge (|u_i^k - \lambda^k| < \theta^k)$
**return** $check$
```

the median $\boldsymbol{\lambda}$ of all cluster means to compute a threshold $\boldsymbol{\theta}$ for model updates. The values of median $\boldsymbol{\lambda}$ and threshold $\boldsymbol{\theta}$ are public, and broadcasted by the server to all clients. Each client i then provides a zero-knowledge proof attesting that their update is within the threshold from the median, i.e., $\text{abs}(\boldsymbol{u_i} - \boldsymbol{\lambda}) < \boldsymbol{\theta}$. This ensures that clients are not performing Byzantine attacks on the central model (item ① in threat model). Users that fail to provide the proof are considered malicious and treated as dropped. The remaining users participate in a round of `zPROBE` secure aggregation and the server obtains the final aggregate result.

4.3.2 `zPROBE` Secure Aggregation

Alg. 7 shows the detailed steps for `zPROBE`'s secure aggregation for n clients consisting of three rounds. In round 1, each client i generates a key pair (sk_i, pk_i), samples a random seed b_i, and performs a key agreement protocol [157] with client j to obtain a shared seed $a_{i,j}$. The seeds are used to generate individual and pairwise masks using a pseudorandom generator (PRG). Each client then creates t-out-of-n Shamir shares (SS) of sk_i and b_i, and sends one share of each to every other client.

In the second round, each client uses the masks generated in round one to compute masked updates according to Eq. 4.1, which are then sent to the server. All clients perform the ZKP authentication protocol described in Sec. 4.2 on their update. This ensures that clients use consistent update values across different steps (item ② in threat model). In addition, each client proves, in zero-knowledge, that their sent value $\boldsymbol{v_i}$ is correctly computed as shown in Alg. 8. Specifically, the circuit that is evaluated in zero-knowledge expands the generated seeds to masks, and computes the masked update using Eq. 4.1. The value of *check* is then opened by the client, and the server verifies that $check = 1$. This ensures that the masks are correctly generated from seeds, and the masked update is correctly computed (item ③ in the threat model). Users that fail to provide the proof are dropped in the next round and their update is not incorporated in aggregation.

We introduce optimizations in Sec 4.3.4 that allow the server to derive a bound q, for the number of model updates to be checked, such that the probability of detecting Byzantine updates is higher than a predefined rate. The server samples q random parameters from client i, and performs the update correctness check (Alg. 8). We note that clients are not motivated to modify the seeds for creating masks, since this results in uncontrollable, out-of-bound errors that can be easily detected by the server (item ④ in threat model). We discuss the effect of using wrong seeds in Section 4.5.2

In round 3, the server performs unmasking by asking for shares of sk_i for dropped users and shares of b_i for surviving users, which are then used to reconstruct the pairwise and individual masks for dropped and surviving users respectively. The server is then able to obtain the aggregate result.

4.3.3 Establishing Robustness

Deriving Dynamic Bounds. We dynamically determine the range for *acceptable* gradients in each iteration, identifying those that don't hinder the central model's convergence. This is based on the assumption that benign updates are majority and harmful Byzantine updates are minority outliers. The median, even in the presence of outliers, acts as a reliable baseline for in-distribution values.

In the secure FL setup, the true value of the individual user updates is not revealed to the server. Calculating the median on the masked user updates is therefore nontrivial since it requires sorting the values which incurs extremely high overheads in secure domain. We circumvent this challenge by forming clusters of users, where our secure aggregation can be used to efficiently compute the average of their updates. The secure aggregation abstracts out the user's individual updates, but reveals the final mean value for each cluster $\{\boldsymbol{\mu_1}, \boldsymbol{\mu_2}, \ldots, \boldsymbol{\mu_c}\}$ to the server. The server can thus easily compute the median ($\boldsymbol{\lambda}$) on the mean of clusters in plaintext.

Using the Central Limit Theorem for Sums, cluster means follow a normal distribution $\boldsymbol{\mu_i} \sim \mathcal{N}(\boldsymbol{\mu}, \frac{1}{\sqrt{n_c}}\boldsymbol{\sigma})$ where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ denote the mean and standard deviation of the original model updates and n_c is the cluster size. We can thus use the standard deviation of the cluster means ($\boldsymbol{\sigma_\mu}$) as a distance metric for marking outlier updates. The distance is measured from the median of means $\boldsymbol{\lambda}$, which serves as an acceptable model update drawn from $\mathcal{N}(\boldsymbol{\mu}, \frac{1}{\sqrt{n_c}}\boldsymbol{\sigma})$. For a given update $\boldsymbol{u_i}$, we investigate Byzantine behavior by checking $|\boldsymbol{u_i} - \boldsymbol{\lambda}| < \boldsymbol{\theta}$, where $\boldsymbol{\theta} = \eta . \boldsymbol{\sigma_\mu} = \frac{\eta}{\sqrt{n_c}}\boldsymbol{\sigma}$. The value of η can be tuned based on cluster size (n_c) and the desired statistical bounds on the distance in terms of the standard deviation of model updates ($\boldsymbol{\sigma}$). Specifically, assuming a higher bound on the portion of malicious users ϕ_{max}, the server can automatically adjust η such that at most $(1 - \phi_{max}) \cdot n$ of the users are marked as benign where n is the total user count.

Secure Robustness Check. We use ZKPs to identify malicious clients that send invalid updates, without compromising clients' privacy. Our ZKP relies on the robustness metrics derived in Sec. 4.3.3, i.e., the median of cluster means $\boldsymbol{\lambda}$ and the threshold $\boldsymbol{\theta}$. Clients ($\mathscr{P}$) prove to the server ($\mathscr{V}$) that their updates comply with the robustness range check.

During the aggregation round in step 1, clients authenticate their updates, and the authenticated value is used in steps 2 and 3. This ensures that consistent values are used across steps and clients can not change their update after learning $\boldsymbol{\lambda}$ and $\boldsymbol{\theta}$ to fit in the robustness

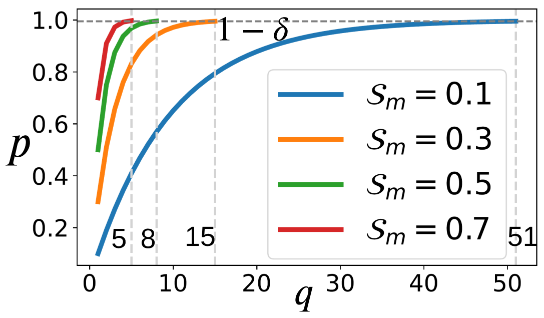


Figure 4.2. Detection probability vs. number of ZKP checks (q). Vertical lines mark the required q values for 99.5% detection rate.

threshold. In step 2, the server makes $\boldsymbol{\lambda}$ and $\boldsymbol{\theta}$ public. Inside ZKP, the clients' updates $\boldsymbol{u_i}$ are used in a Boolean circuit determining if $|\boldsymbol{u_i} - \boldsymbol{\lambda}| < \boldsymbol{\theta}$ as outlined in Alg. 9. Invalid model updates failing the range check are excluded from the final aggregation.

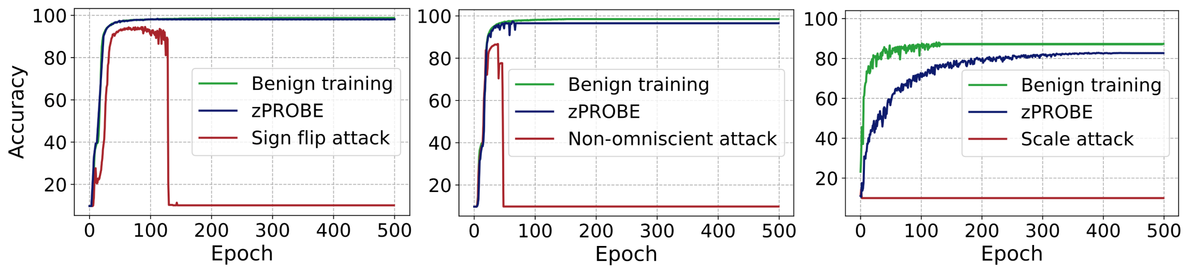
4.3.4 Probabilistic Optimizations

This section provides statistical bounds on the number of required checks to accurately detect malicious clients. Using the derived bounds, we optimize our framework for minimum overhead, thus ensuring scalability to large models.

Malicious clients can compromise their update, by sending updates with distance margins larger than the tolerable threshold θ, or sending incorrect masked updates (Eq. 4.1). Assume that a portion of model updates $\mathscr{S}_m$, are compromised. The probability of detecting a malicious update is equivalent to finding at least one compromised parameter gradient:

$$p = 1 - \binom{l \cdot (1 - \mathscr{S}_m)}{q} \Big/ \binom{l}{q}, \tag{4.2}$$

where l is the total model parameter updates, and q denotes the number of per-user ZKP checks on model updates. The above formulation confirms that it is indeed not necessary to perform ZKP checks on all parameter updates within the model. Rather, q can be easily computed via Eq. 4.2, such that the probability of detecting a compromised update is higher than a predefined rate: $p > 1 - \delta$. Fig. 4.2 shows the probability of detecting malicious users versus number of



(a) MNIST + Sign Flip (b) FMNIST + Non-omniscient (c) CIFAR-10 + Scaling

Figure 4.3. Test accuracy vs. FL training epochs for different attacks and benchmarks. Each plot shows the benign training (green), Byzantine training without defense (maroon), and Byzantine training with `zPROBE` defense.

ZKP checks for a model with $l = 60K$ parameters. As seen, `zPROBE` guarantees a failure rate lower than $\delta = 0.005$ with very few ZKP checks. Note that malicious users are incentivized to attack a high portion of updates to increase their effect on the aggregated model's accuracy. We leverage Eq. 4.2 to derive the required number of correctness and robustness checks as described in Alg. 8 and Alg. 9. For each check, the server computes the bound q, then samples q random indices from model parameters for each client. The clients then provide ZKPs for the selected set of parameter indices.

4.4 Experiments

4.4.1 Experimental Setup

Table 4.1. Comparison of `zPROBE` and previous robust FL aggregators for non-IID data, with accuracy reported across 10 runs.

Aggregator	IID	Non-IID
AVG	93.2 ± 0.2	92.7 ± 0.3
KRUM	91.6 ± 0.3	53.1 ± 3.9
CM	91.9 ± 0.2	78.6 ± 3.1
CClip	93.0 ± 0.2	91.2 ± 0.5
RFA	93.2 ± 0.2	92.6 ± 0.2
`zPROBE`	99.0 ± 0.0	98.6 ± 0.4

We now provide details about the benchmarked models, datasets, and defense implementation.

Dataset and Models. We consider three benchmarks commonly studied by prior work in secure FL. Our first benchmark is a variant of LeNet5 [158] trained on the MNIST dataset [159], with 2 convolution and 3 fully-connected layers, totaling $42K$ parameters. Our second benchmark is the Fashion-MNIST (F-MNIST) dataset [160] trained on the LeNet5 architecture with $60K$ parameters. Finally, to showcase the scalability of our approach, we evaluate ResNet-20 [161] with $273K$ parameters trained on the CIFAR-10 dataset [162] which is among the biggest benchmarks studied in the secure FL literature [163]. Table 4.2 encloses the training hyperparameters for all models.

Table 4.2. Training hyperparameters.

Benchmark	# Clients	LR	# Epochs	Batch size (per user)
MNIST (IID) + LeNet5	50	0.01	500	12800 (256)
MNIST (non-IID) + LeNet5	25	0.01	500	800 (32)
F-MNIST (IID) + LeNet5	50	0.01	500	12800 (256)[†]
CIFAR-10 (IID) + ResNet-20	50	0.05	500	12800 (256)
CIFAR-100 (IID) + ResNet-18	50	0.05	500	12800 (256)

[†]When varying the number of clients, we keep the total batch size as 12800 and scale the per user batch size accordingly.

Implementation and Configuration. zPROBE defense is implemented in Python and integrated in PyTorch to enable model training. We use the EMP-Toolkit [164] for implementation of zero-knowledge proofs. We run all experiments on a 128GB RAM, AMD Ryzen 3990X CPU desktop. All reported runtimes are averaged over 100 trials.

Byzantine Attacks. We assume 25% of the clients are Byzantine, which is a common assumption in the literature [147]. Malicious users alter a portion $\mathscr{S}_m$ of benign model updates and masks according to a Byzantine attack scenario. We show the effectiveness of our robustness checks against three commonly used Byzantine attacks. Here $\mathscr{U}_m$ denotes the malicious updates, where $|\mathscr{U}_m| = \mathscr{S}_m \cdot l$ and l is the total number of model updates.

- *Sign Flip* [145]. Malicious client flips the sign of the update: $u = -\kappa.u, \kappa > 0 \ (\forall u \in \mathscr{U}_m)$

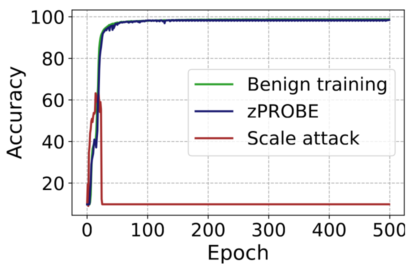
- *Scaling* [146]. Malicious client scales the local gradients to increase the influence on the global model: $u = \kappa.u, \kappa > 0 \;\; (\forall u \in \mathscr{U}_m)$
- *Non-omniscient attack* [147]. Malicious clients construct their Byzantine update by adding a scaled Gaussian noise to their original update with mean μ and standard deviation σ: $u = \mu - \kappa.\sigma \;\; (\forall u \in \mathscr{U}_m)$

Baseline Defenses. We present comparisons with prior work on robust and private FL, i.e., BREA [7] and EIFFeL [8]. While `zPROBE` is able to implement popular defenses based on rank-based statistics, EIFFeL is limited to static thresholds and requires access to clean public datasets. BREA implements multi-Krum [165], but leaks pairwise distances of clients to the server. `zPROBE` achieves lower computation complexity compared to both works and higher accuracy[1] compared to EIFFeL. We also benchmark a commonly used aggregator which uses only the median of cluster means, and show that it results in drastic loss of accuracy compared to `zPROBE`. Additionally, we evaluate `zPROBE` when the training data is non-IID and show our adaptive bounds outperform the state-of-the-art defense in [166][2].

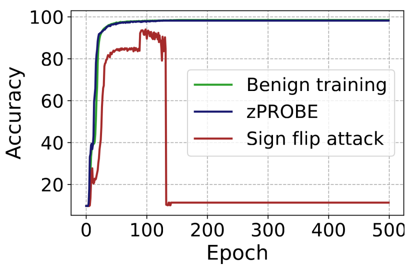
4.4.2 Defense Performance

IID Training Data. We evaluate `zPROBE` on various benchmarks using $n = 50$ clients picked for a training round, randomly grouped into $c = 7$ clusters. In Section 4.5.6 we present evaluations with different number of clients between 30 and 200. Consistent with prior work [147], we assume malicious users compromise all model updates to maximize the degradation of the central model's accuracy. Fig. 4.3 demonstrates the convergence behavior of the FL scheme in the presence of Byzantine users with and without `zPROBE` defense. As seen, `zPROBE` successfully eliminates the effect of malicious model updates and recovers the ground-truth accuracy. We show evaluations of `zPROBE` accuracy on other variants of the dataset and attack in Fig. 4.4.

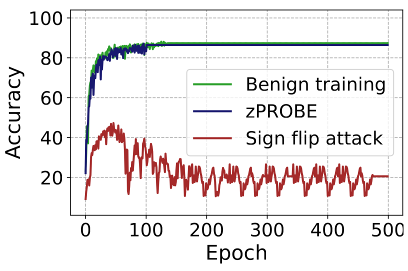
[1]Raw accuracy numbers are not reported for BREA, therefore, direct comparison is not possible.
[2]Note that this work focuses on plaintext robust training and does not provide secure aggregation.



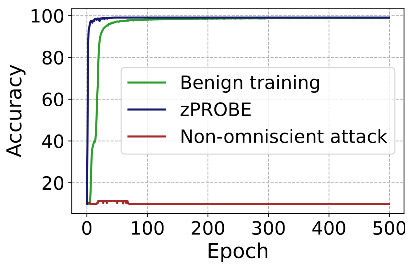
(a) MNIST + Scale attack



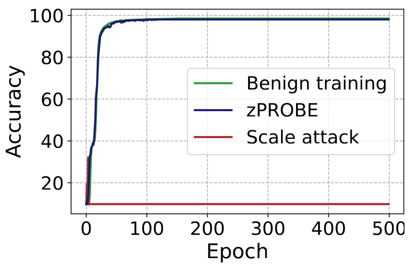
(b) F-MNIST + Sign flip attack



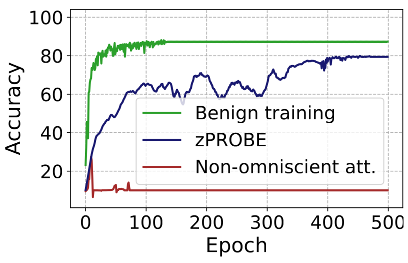
(c) CIFAR-10 + Sign flip attack



(d) MNIST + Non-omniscient attack



(e) F-MNIST + Scale attack



(f) CIFAR-10 + Non-omniscient

Figure 4.4. Test accuracy as a function of FL training epochs for different attacks and benchmarks. Each plot shows the benign training (green), Byzantine training without defense (maroon), and Byzantine training in the presence of `zPROBE` defense.

On the MNIST benchmark, the byzantine attacks cause the central model's accuracy to reduce to nearly random guess (10.2%-11.2%), without any defense. `zPROBE` successfully thwarts the malicious updates, recovering benign accuracy within 0.0%-0.6% margin. On F-MNIST, we recover the original $\sim 88\%$ drop of accuracy caused by the attacks to 0%-2% drop. Finally, on CIFAR-10, the gap between benign training and the attacked model is reduced from 45%- 90% to only 3%-7%. Compared to EIFFeL [8], `zPROBE` achieves 1.2%, 0.5%, and 2.8% higher accuracy when evaluated on the same attack applied to MNIST, FMNIST, and CIFAR-10, respectively.

Non-IID Training Data. Most recently, [166] show user clustering over existing robust aggregation methods can adapt them to heterogeneous (non-IID) data. We follow their training setup and hyperparameters to distribute the MNIST dataset unevenly across 25 users. As shown in Tab. 4.1, `zPROBE` defense outperforms the accuracies obtained by the various defenses evaluated in [166]. This performance boost we believe can be attributed to 1) the use of rank-based statistics to establish dynamic thresholds, and 2) the use of all benign gradients in the aggregation, rather than replacing all values with a robust aggregator, e.g., as in KRUM.

4.4.3 Runtime and Complexity Analysis

Tab. 4.3 summarizes the total runtime for clients in `zPROBE` for one round of federated training with $n = 50$, $c = 7$, and $\mathscr{S}_m = 0.3$ across different benchmarks. We use the secure aggregation protocol of [6] as our baseline, which does not provide security against malicious clients or robustness against Byzantine attacks.

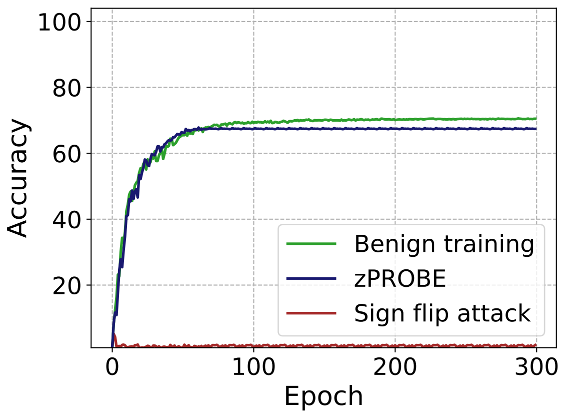
Table 4.3. `zPROBE` runtime vs. the baseline secure aggregation of [6] with no support for Byzantine clients.

Dataset	Baseline (ms)	`zPROBE` (ms)
MNIST	208.0	444.7
F-MNIST	214.4	452.9
CIFAR-10	231.2	461.2
CIFAR-100	1796.3	2314.2

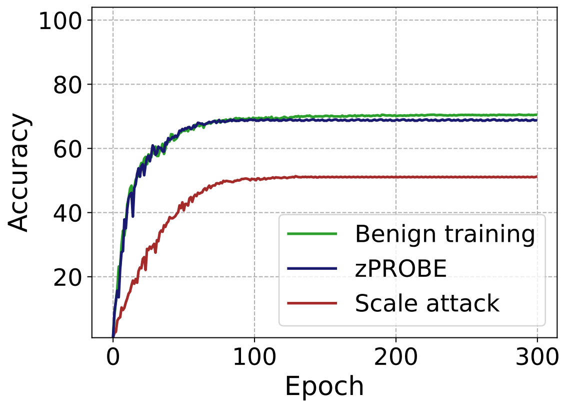
Tab. 4.4 summarizes the runtime of `zPROBE` versus the portion of attacked model updates. By decreasing $\mathscr{S}_m$, `zPROBE` requires more checks to detect the outlier gradients as outlined in Eq. 4.2. Nevertheless, due to the optimizations in `zPROBE` robustness and correctness checks, we are still able to maintain practical runtime and sublinear growth with respect to number of ZKP checks necessary.

Table 4.4. `zPROBE` performance for LeNet5 on F-MNIST vs. the portion of Byzantine model updates ($\mathscr{S}_m$).

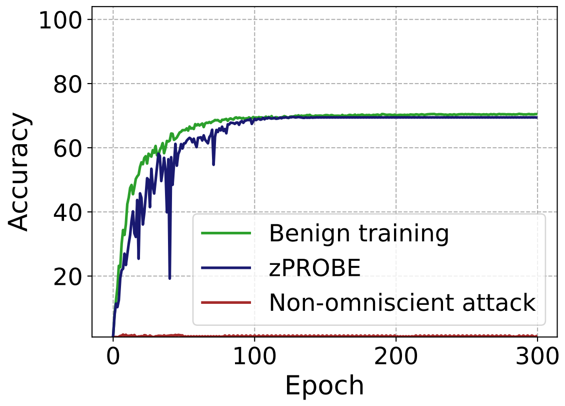
	$\mathscr{S}_m$				
	0.1	0.3	0.5	0.7	1.0
# ZKP Checks	51	15	8	5	1
`zPROBE` Runtime (ms)	777.9	452.9	372.6	349.6	316.5



(a) CIFAR-100 + Sign flip attack



(b) CIFAR-100 + Scale attack



(c) CIFAR-100 + Non-omniscient attack

Figure 4.5. Test accuracy as a function of FL training epochs for different attacks and using CIFAR-100 dataset. Each plot shows the benign training (green), Byzantine training without defense (maroon), and Byzantine training in the presence of `zPROBE` defense.

4.4.4 Evaluation on large dataset

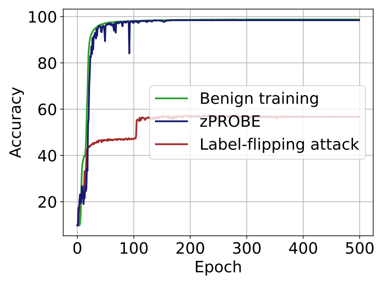
We perform the evaluation on the real-world dataset CIFAR-100. The CIFAR-100 dataset is a collection of 60,000 32x32 color images divided into 100 classes, each containing 600 images. It's an extension of the CIFAR-10 dataset but with more categories. Each class in CIFAR-100 is further grouped into 20 superclasses, providing a more detailed and challenging dataset for image classification tasks. In this experiment, CIFAR-100 is IID, and we use a slightly different setting because the dataset has 100 classes, and each class has 600 images. Instead of distributing unique data to each user evenly, including malicious users, we randomly distribute 25% of the data to each user. The graph in Fig. 4.5 demonstrates the convergence of the FL scheme when Byzantine users are present, both with and without `zPROBE` protection. Note that we train the CIFAR-100 for 300 epochs. From Fig. 4.5, we can observe that the accuracy gap between the benign model and the attacked model is reduced from 17%- 69% to only 0.4%-0.9% for the considered three types of attacks. Therefore, `zPROBE` is constantly successful in mitigating the impact of malicious model updates and restoring the accuracy of the true values.

4.4.5 Evaluation on additional Byzantine attacks

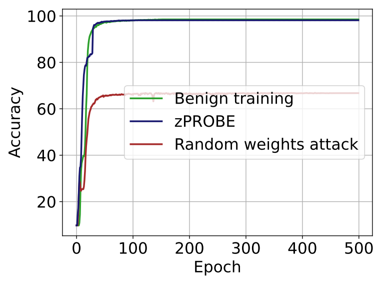
In this section, we performance additional Byzantine attacks from the state-of-the-art works. Especially, we use the following two attack methods:

- *Label-flipping attack [167]* Malicious clients randomly flip the labels of the adversarial attacks and send their Byzantine update with incorrect information.
- *Random weights attack [168]* Malicious clients use randomly generated model updates as Byzantine updates.

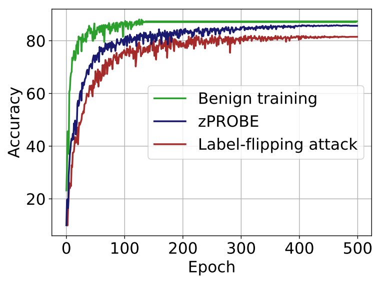
We evaluate all of the datasets on these additional attacks, and Fig. 4.6 shows the performance results for MNIST, F-MNIST and CIFAR-10 datasets. As we can observe from Fig. 4.6, the accuracy gap between the benign model and the attacked model is reduced from 8%- 70% to only 0.4%-0.5%. Fig. 4.7 also shows the performance of CIFAR-100 dataset. We can observe that `zPROBE` is able to recover the accuracy from 66% to the accuracy that benign training offers for label-flipping attack and recover the accuracy from around 0% to an acceptable accuracy for random weights attacks. Again, `zPROBE` is shown to be efficient in mitigating the impact of malicious model updates and restoring the training accuracy. Evaluations on additional Byzantine attacks and larger datasets corroborate our conclusion, suggesting our framework can defend against more attacks with large datasets.



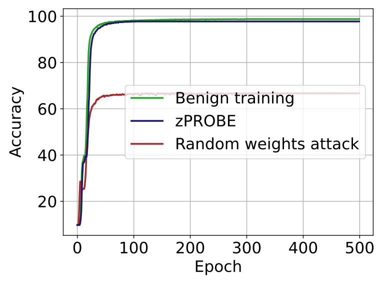
(a) *MNIST + Label flip attack*



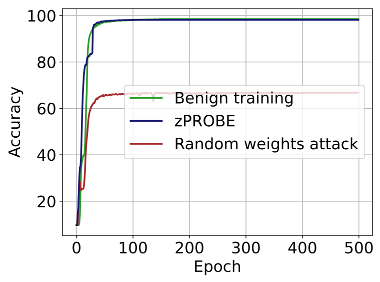
(b) *F-MNIST + Random weights*



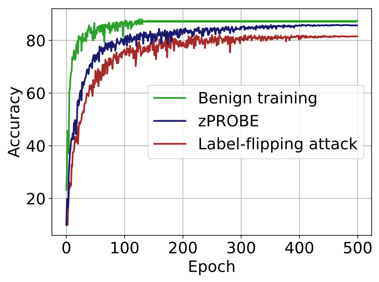
(c) *CIFAR-10 + Label flip attack*



(d) *MNIST + Label flip attack*



(e) *F-MNIST + Random weights*



(f) *CIFAR-10 + Label flip attack*

Figure 4.6. *Test accuracy as a function of FL training epochs for additional attacks and benchmarks. Each plot shows benign training (green), Byzantine training without defense (maroon), and Byzantine training in the presence of* `zPROBE` *defense.*

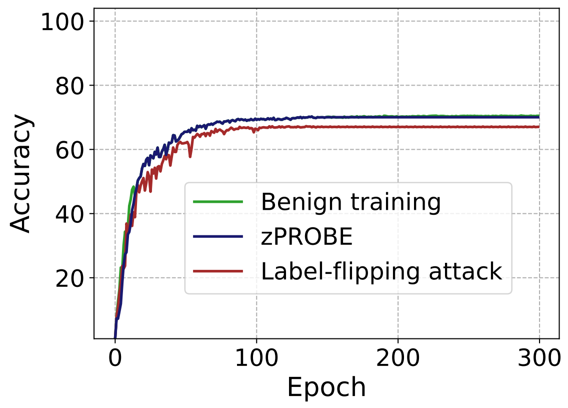
4.4.6 Attribute inference attack

In federated learning, an attribute inference attack aims to deduce sensitive data attributes from participant nodes without direct data access. In our work, we show effective protection against gradient inversion attacks, [169] shows that attribute inference attacks result in similar lapses of effectiveness as the scale is increased, following the same trend as gradient inversion attacks. Therefore, `zPROBE` *is able to defend against attribute inference attacks.*

4.5 Sensitivity Analysis

4.5.1 Effect of Number of Clients on `zPROBE` Runtime

Tab. 4.5 shows the effect of increasing the number of clients on the performance on `zPROBE`. For these experiments, results are gathered on the F-MNIST dataset, with $c = 7$ and $|\mathscr{S}_m| = 0.3$. As seen, although exceeding sub-second performance as the number of clients



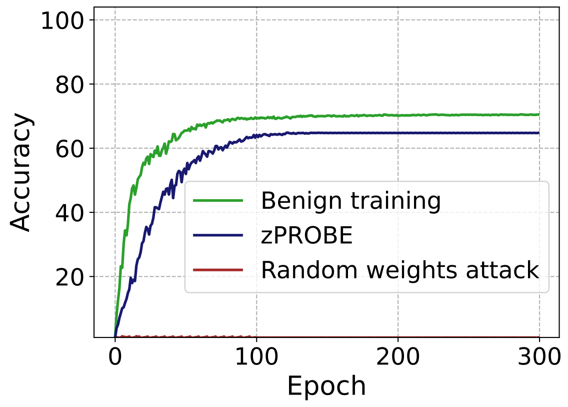


Figure 4.7. *Test accuracy as a function of FL training epochs for additional attacks and CIFAR-100 benchmarks (Top): CIFAR-100 + Label-flipping attack, and (Bottom) CIFAR-100 + Random weights attack.*

scales up, `zPROBE` maintains sublinear growth in runtime with respect to number of clients. In Tab. 4.3, we can see that as the underlying model gets much larger (growing $\sim 4\times$ in size from the MNIST to CIFAR-10 tasks) `zPROBE` overhead grows a negligible amount. This is a strong indicator of the scalability of our proposed secure aggregation. Alongside this, the probabilistic optimizations explained in Sec. 4.3.4 become more beneficial as the model size increases. Compared to a naive implementation where $1-\mathscr{S}_m$ parameters are checked, we achieve a speedup of 3 orders of magnitude in client and server runtime.

We present a detailed runtime breakdown of various `zPROBE` components in Fig. 4.8, using the CIFAR-10 dataset, ResNet-20 architecture, $n = 50$, $c = 7$, and $\mathscr{S}_m = 0.3$. Step 2 exhibits low overhead, even with 30% Byzantine model updates. The most significant increase in overhead, in terms of percentage, occurs in Step 3 (R2), where masked updates are checked for

Table 4.5. Runtime of `zPROBE` over varying number of clients

# Clients	`zPROBE` Runtime (ms)
30	298.4
40	369.7
50	452.9
70	598.8
100	828.6
200	1620.4

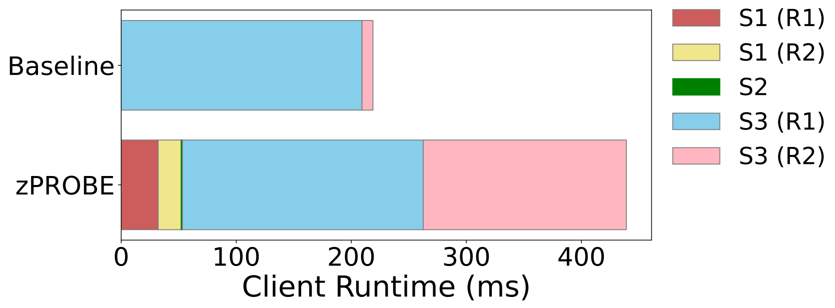


Figure 4.8. Runtime breakdown for CIFAR-10, corresponding to rounds (R) from Alg. 7 and steps (S) from Fig. 4.1.

correctness (Alg. 7 Round 2 and Alg. 8). In terms of communication overhead, `zPROBE` requires only 2.1MB and 4.4MB of client and server communication, respectively, for a CIFAR-10 aggregation round. Overall, `zPROBE` offers efficient, privacy-preserving, and robust federated learning performance across all benchmark tests.

`zPROBE` Complexity. In this section we present the complexity analysis of `zPROBE` runtime with respect to number of clients n (with $k = \log n$) and model size l.

• Client: Each client computation consists of performing key agreements with $O(k)$, generating pairwise masks with $O(k \cdot l)$, creating t-out-of-k Shamir shares with $O(k^2)$, performing correctness checks of Alg. 8 with $O(k \cdot l)$, and performing robustness checks of Alg. 9 with $O(l)$. The complexity of client compute is therefore $O(\log^2 n + l \cdot \log n)$.

• Server. The server computation consists of reconstructing t-out-of-k shamir shares with $O(n \cdot k^2)$, generating pairwise masks for dropped out clients with $O(n \cdot k \cdot l)$, performing correctness checks of Alg. 8 with $O(n \cdot l)$, and performing robustness checks of Alg. 9 with $O(n \cdot l)$. The overall complexity of server compute is thus $O(n \cdot \log^2 n + n \cdot l \cdot \log n)$.

We are unable to directly compare zPROBE's runtime numbers with previous private and robust FL methods since their implementations are not publicly available. Instead, Tab. 4.6 presents a complexity comparison between zPROBE, BREA [7], and EIFFeL [8] with respect to number of clients n (with $k = \log n$), model size l, and number of malicious clients m. zPROBE enjoys a lower computational complexity compared to both prior art for client and server. Specifically, the client runtime is quadratic and linear with number of clients in BREA and EIFFeL respectively, whereas logarithmic in zPROBE.

Table 4.6. Runtime complexity of zPROBE vs. prior works BREA [7] and EIFFeL [8].

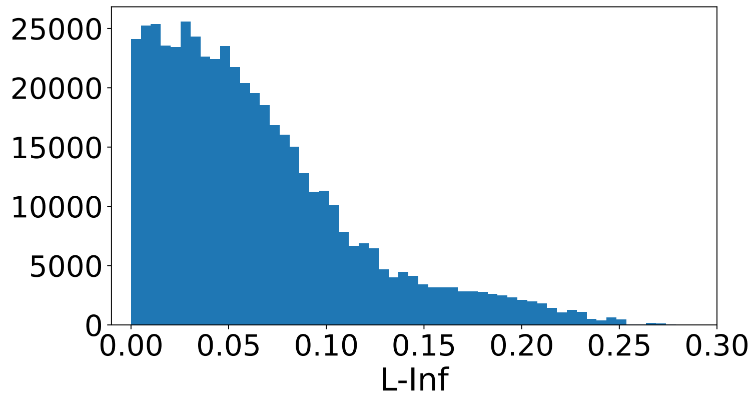
	Client	Server
BREA	$O(n^2 l + nlk^2)$	$O((n^3 + nl)k^2 \cdot \log(k))$
EIFFeL	$O(mnl)$	$O((n+l)nk^2 \cdot \log(k) + m.l.\min(n, m^2))$
zPROBE	$O(k^2 + kl)$	$O(nk^2 + nlk)$

4.5.2 Malicious Seed Modification

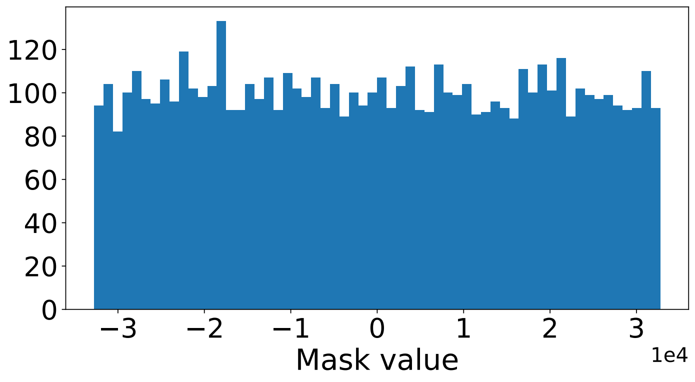
Fig. 4.9(a) displays the histogram of benign gradient L_∞ norms across 100 CIFAR-10 users, predominantly within $[0, 0.25]$. Seeds from a pseudorandom generator (PRG) create masks, impervious to manipulation by malicious users. Fig. 4.9(b) illustrates mask value histograms from varying seeds over 10,000 instances, showing potential drastic fluctuations between -3×10^4 and 3×10^4. Consequently, random seed alterations by malicious users in mask generation are detectable due to significant gradient errors, thus discouraging such actions in our threat model.

4.5.3 Effect of Aggregation Method on Accuracy

zPROBE uses the median of averaged updates from user clusters to distinguish benign from Byzantine updates. An alternative method is the median of cluster means, bypassing per-user checks. Fig. 4.10 compares this method's test accuracy against zPROBE during training. The baseline, using the median of cluster means, shows significant accuracy loss compared to



(a)



(b)

Figure 4.9. Histogram of (a) ResNet-20 gradient norms observed during training on CIFAR-10, and (b) mask values when changing the random seed.

`zPROBE`. This is because it may include Byzantine workers and loses valuable information in benign updates by solely relying on the median.

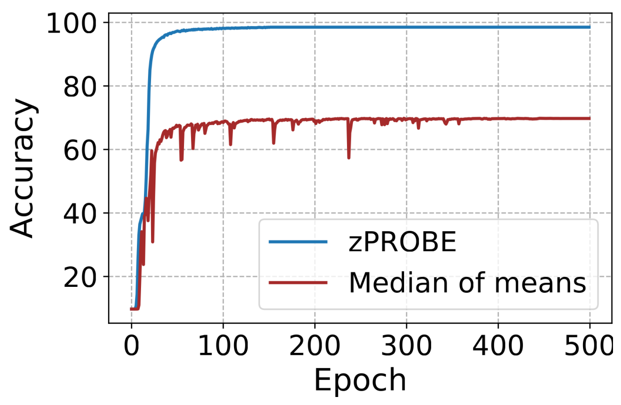
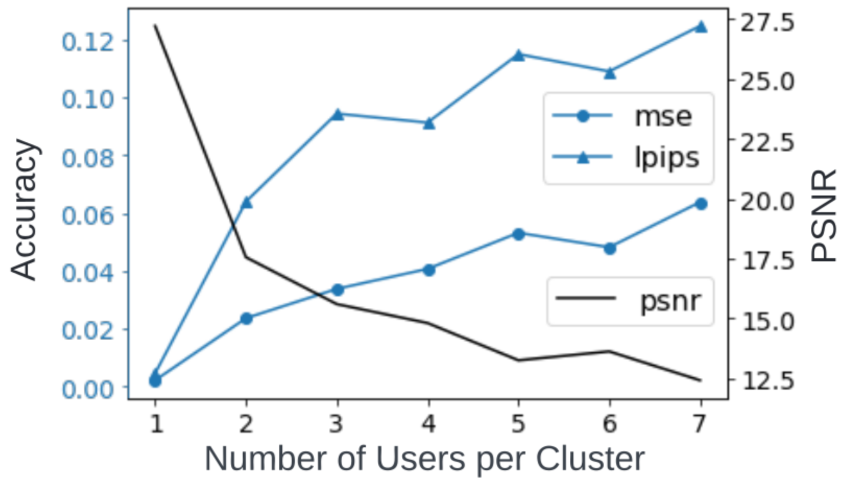


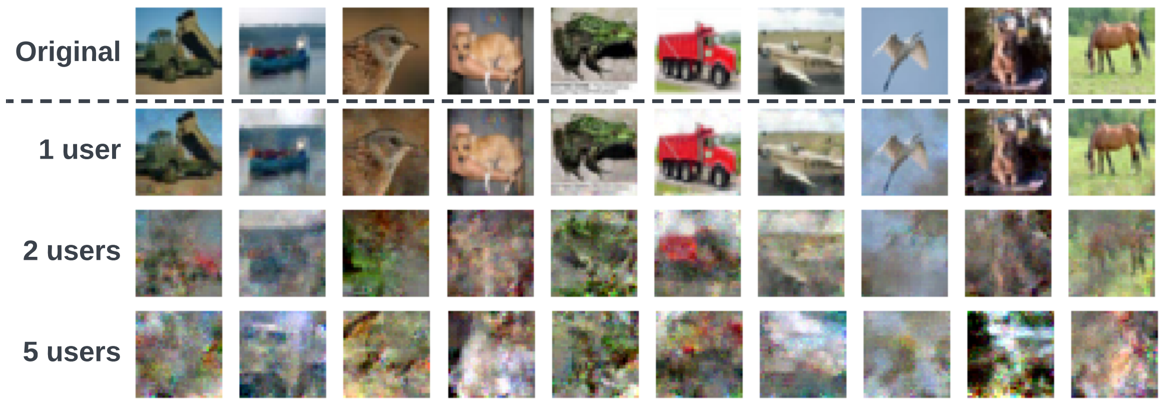
Figure 4.10. Test accuracy of `zPROBE` compared with an aggregation methodology that uses the median of cluster means.

4.5.4 Effect of Cluster Size on Inversion Attack

Fig. 4.11 shows the effect of cluster size on gradient inversion attacks. In Fig 4.11(a) we show the effectiveness of the attack for different cluster sizes. Fig 4.11(b) represents the reconstruction results from user data for different number of users participating in the aggregation round.



(a)



(b)

Figure 4.11. Performance of gradient inversion attacks for different cluster sizes.

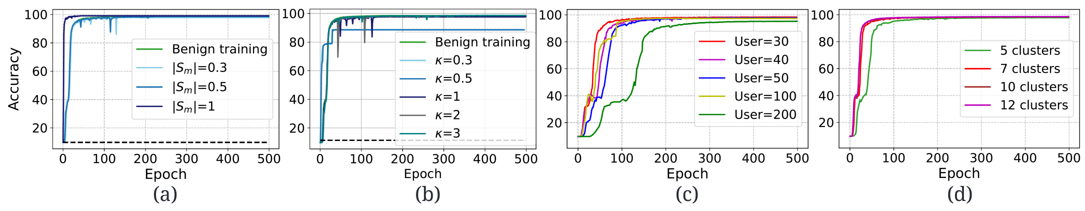


Figure 4.12. Ablation studies on zPROBE defense performance with varying (a) portion of compromised gradients, (b) attack magnitude, (c) number of clients, and (d) number of user clusters. The dashed line in (a), (b) corresponds to the highest test accuracy obtained during training when no defense is applied.

4.5.5 zPROBE Test Accuracy

Fig. 4.4 shows the test accuracy of zPROBE in face of different variations of Byzantine attacks and datasets. The dataset is distributed evenly (IID) among $n = 50$ clients. The server randomly clusters users into $c = 7$ groups during each training round. We assume malicious users compromise all model updates $|\mathscr{S}_m| = 1$ to maximize the accuracy degradation.

4.5.6 Discussion

We perform a sensitivity analysis to various attack parameters and FL configurations on F-MNIST. Number of clients is set to $n = 50$ with $c = 7$ clusters, unless otherwise noted. Sign flip attack [145] is applied to all model updates with $\kappa = 5$. As shown, zPROBE is largely robust to changes in the underlying attack or training configuration, consistently recovering the central models' accuracy.

Portion of Compromised Updates $\mathscr{S}_m$. We vary the portion of Byzantine model updates ($\mathscr{S}_m$) and show the accuracy of the central model with and without zPROBE robustness checks in Fig. 4.12(a). Even when only a small portion of model updates are malicious, zPROBE's outlier detection can successfully recover the accuracy from random guess (10%) to 97.9%. Byzantine workers try to reduce the central model's accuracy by attacking multiple model updates. Without defense, this drops accuracy by 89.7%. However, zPROBE successfully maintains the central model's accuracy with less than 0.5% error.

Attack Magnitude. We control the magnitude of the perturbation applied to model updates by changing the parameter κ in various Byzantine attack scenarios. Fig. 4.12(b) shows the effect of the attack magnitude on the central model's accuracy and zPROBE's defense performance. As seen, a higher perturbation is easier to detect using our median-based robustness check. The Byzantine attack can cause an accuracy drop of $\sim 88\%$ when no robustness check is applied. However, zPROBE can largely recover the accuracy degradation, reducing the accuracy loss to 0.2%-9.8%.

Defense Parameter. *Our robustness check uses a threshold θ based on the standard deviation (SD) of cluster mean updates, $\theta = \eta.\sigma_{\mu}$. By adjusting η, we control the outlier detection's strictness. For instance, setting $\frac{\eta}{\sqrt{n}} = 2$ allows approximately 95.4% of update values to pass, following the normal distribution. We explored how different η values affect robustness, and discovered that a smaller η effectively eliminates all malicious updates, similar to the Trimmed-mean defense strategy.*

Aggregation Strategy. In Fig. 4.1, zPROBE employs the median of per-cluster means to set a threshold in step 2 for filtering malicious clients. An alternative robust aggregation approach directly uses the median value to update the global model, skipping steps 2 and 3 of zPROBE. However, this simple median-based approach ignores valuable updates from benign users, causing a severe accuracy drop of 28.6% compared to zPROBE, which includes all passing gradients. Fig. 4.10 illustrates this comparison.

Number of Clients (n). Fig. 4.12(c) demonstrates that zPROBE's training convergence is unaffected by the number of clients (n ranging from 30 to 200) selected per aggregation round in Federated Learning (FL). This selection aligns with real-world FL practices where only a fraction of clients are chosen each round. Notably, zPROBE maintains consistent model accuracy and scales effectively up to 200 clients, a significant number for robust and private FL studies.

Fig. 4.12(d) shows the effect of number of clusters on accuracy. The performance of zPROBE is largely independent of the number of clusters, showing less than 0.18% variation for different c while the increase in latency is less than 8%. The number of clusters can therefore be selected freely such that user privacy is ensured.

User Dropout. `zPROBE` secure aggregation supports user dropouts, i.e., when a user is disconnected amidst training iterations and/or in between `zPROBE` steps (see Fig. 4.1). Fig 4.13 shows the effect of random user dropouts on `zPROBE` defense. As shown, the fluctuations in the central model's test accuracy are negligible ($< 0.13\%$). The robustness of `zPROBE` aggregation protocol to user dropouts is intuitive since the remaining users can carry on the training.

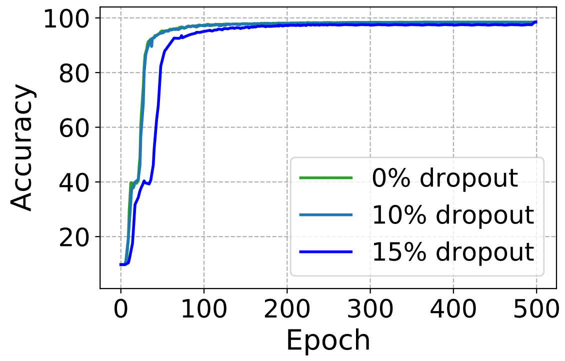


Figure 4.13. The effect of user dropout on defense.

Convergence. *Prior work [170] provides convergence proof for a median of means aggregation, which is how `zPROBE` aggregates user updates and establishes robustness thresholds. `zPROBE` aggregation slightly modifies the median of means but is range-bounded and can be proved to converge using the same analysis. Under the same probabilistic framework presented in [170], `zPROBE` stays within the thresholds of convergence, as the percentage of malicious clients, 25%, is less than $(n-1)/2$. In this scenario, n is the total number of clients. By staying within this bound, we are able to adopt the theoretical guarantee of convergence of the median of means aggregation.*

4.5.7 Security Analysis

`zPROBE`'s security is validated under the Universal Composability (UC) framework, which ensures that security properties of individual cryptographic components are maintained in a combined end-to-end system. This framework guarantees privacy in `zPROBE` through Zero-Knowledge Proofs (ZKPs), ensuring no user information leakage while enabling robustness

and consistency checks. The secure aggregation used in updating the global model ensures that the server learns only the aggregate value, not individual updates. `zPROBE`*'s security approach aligns with the analyses in [6], thus conforming to UC framework standards. The probability of not detecting malicious users is capped at 0.5%, as detailed in Sec. 4.3. Additionally, it is assumed that both clients and server are aware of the model's nature.*

4.6 Future Work

This section outlines several potential directions for future research:

- **Efficient Zero-Knowledge Proof Constructions**: Further exploration of more efficient zero-knowledge proof constructions to reduce computational overhead for both clients and the server.
- **Clustering Strategies**: Investigate the impact of different clustering strategies on the robustness and privacy guarantees of the zPROBE framework.
- **Handling Complex Byzantine Attack Scenarios**: Extend the zPROBE framework to manage more complex Byzantine attack scenarios, including coordinated attacks by multiple malicious clients.
- **Theoretical Analysis and Convergence Guarantees**: Analyze the theoretical properties and convergence guarantees of the zPROBE approach for federated learning, beyond the empirical evaluations presented in this paper.
- **Applicability to Other Distributed Learning Settings**: Explore the applicability of the zPROBE approach to other distributed learning settings beyond federated learning, such as decentralized or peer-to-peer learning.

4.7 Conclusion

This chapter introduces `zPROBE`, a novel framework for private, robust Federated Learning (FL) in settings with potentially malicious clients. `zPROBE` ensures client compliance and robustness in model updates using zero-knowledge proofs and secret sharing, without compromising privacy. "We highlight the effectiveness of `zPROBE` across seminal computer vision benchmarks, including real-world datasets and various attack methods, but reiterate the fact that our approach is not limited by scale. `zPROBE` can handle even the most advanced computer vision FL tasks, while still maintaining the same levels of privacy and robustness. `zPROBE` presents a paradigm shift from previous work by providing a private robustness defense relying on rank-based statistics with cost that grows sublinearly with respect to number of clients.

4.8 Acknowledgements

This chapter is a reprint of the published material in Z. Ghodsi, M. Javaheripi, N. Sheybani, X. Zhang, K. Huang, and F. Koushanfar, "zPROBE: Zero peek robustness checks for federated learning," in Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 4860-4870, 2023. The dissertation author was the (co)primary investigator and author of these papers.

Bibliography

[1] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E Lauter, and Farinaz Koushanfar. Xonn: Xnor-based oblivious deep neural network inference. In *USENIX Security*, 2019.

[2] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 325–342, 2020.

[3] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[4] Qian Lou, Yilin Shen, Hongxia Jin, and Lei Jiang. {SAFEN}et: A secure, accurate and fast neural network inference. In *International Conference on Learning Representations*, 2021.

[5] Qian Lou, Bian Song, and Lei Jiang. Autoprivacy: Automated layer-wise parameter selection for secure neural network inference. In *Advances in Neural Information Processing Systems*, 2020.

[6] James Henry Bell, Kallista A Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure single-server aggregation with (poly) logarithmic overhead. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1253–1269, 2020.

[7] Jinhyun So, Başak Güler, and A Salman Avestimehr. Byzantine-resilient secure federated learning. *IEEE Journal on Selected Areas in Communications*, 2020.

[8] Amrita Roy Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. Eiffel: Ensuring integrity for federated learning. *arXiv preprint arXiv:2112.12727*, 2021.

[9] Huili Chen, Xinqiao Zhang, Ke Huang, and Farinaz Koushanfar. Adatest: Reinforcement learning and adaptive sampling for on-chip hardware trojan detection. *arXiv preprint arXiv:2204.06117*, 2022.

[10] Yingqi Liu, Wen-Chuan Lee, Guanhong Tao, Shiqing Ma, Yousra Aafer, and Xiangyu Zhang. Abs: Scanning neural networks for back-doors by artificial brain stimulation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1265–1282, 2019.

[11] Jiazhu Dai, Chuanshuai Chen, and Yufeng Li. A backdoor attack against lstm-based text classification systems. *IEEE Access*, 7:138872–138878, 2019.

[12] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

[13] Loc Truong, Chace Jones, Brian Hutchinson, Andrew August, Brenda Praggastis, Robert Jasper, Nicole Nichols, and Aaron Tuor. Systematic evaluation of backdoor data poisoning attacks on image classifiers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 788–789, 2020.

[14] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[15] Yunchun Zhang, Fan Feng, Zikun Liao, Zixuan Li, and Shaowen Yao. Universal backdoor attack on deep neural networks for malware detection. *Applied Soft Computing*, 143:110389, 2023.

[16] Pengzhou Cheng, Zongru Wu, Wei Du, and Gongshen Liu. Backdoor attacks and countermeasures in natural language processing models: A comprehensive security review. *arXiv preprint arXiv:2309.06055*, 2023.

[17] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 691–706. IEEE, 2019.

[18] Nicola Rieke, Jonny Hancox, Wenqi Li, Fausto Milletari, Holger R Roth, Shadi Albarqouni, Spyridon Bakas, Mathieu N Galtier, Bennett A Landman, Klaus Maier-Hein, et al. The future of digital health with federated learning. *NPJ digital medicine*, 3(1):1–7, 2020.

[19] Wensi Yang, Yuhang Zhang, Kejiang Ye, Li Li, and Cheng-Zhong Xu. Ffd: A federated learning based method for credit card fraud detection. In *International conference on big data*, pages 18–32. Springer, 2019.

[20] Bita Darvish Rouhani, Mohammad Samragh, Mojan Javaheripi, Tara Javidi, and Farinaz Koushanfar. Deepfense: Online accelerated defense against adversarial deep learning. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[21] Emanuele Del Sozzo, Davide Conficconi, Marco D Santambrogio, and Kentaro Sano. Senju: A framework for the design of highly parallel fpga-based iterative stencil loop

accelerators. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 233–233, 2023.

[22] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 707–723. IEEE, 2019.

[23] Wenbo Guo, Lun Wang, Xinyu Xing, Min Du, and Dawn Song. Tabor: A highly accurate approach to inspecting and restoring trojan backdoors in ai systems. *arXiv preprint arXiv:1908.01763*, 2019.

[24] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering. *arXiv preprint arXiv:1811.03728*, 2018.

[25] Steffen Eger, Gözde Gül Şahin, Andreas Rücklé, Ji-Ung Lee, Claudia Schulz, Mohsen Mesgar, Krishnkant Swarnkar, Edwin Simpson, and Iryna Gurevych. Text processing like humans do: Visually attacking and shielding nlp systems. *arXiv preprint arXiv:1903.11508*, 2019.

[26] Yuan Zang, Fanchao Qi, Chenghao Yang, Zhiyuan Liu, Meng Zhang, Qun Liu, and Maosong Sun. Word-level textual adversarial attacking as combinatorial optimization. *arXiv preprint arXiv:1910.12196*, 2019.

[27] Wee Chung Gan and Hwee Tou Ng. Improving the robustness of question answering systems to question paraphrasing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6065–6075, 2019.

[28] Prashanth Vijayaraghavan and Deb Roy. Generating black-box adversarial examples for text classifiers using a deep reinforced model. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 711–726. Springer, 2019.

[29] Bin Liang, Hongcheng Li, Miaoqiang Su, Pan Bian, Xirong Li, and Wenchang Shi. Deep text classification can be fooled. *arXiv preprint arXiv:1704.08006*, 2017.

[30] Huili Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. Deepinspect: A black-box trojan detection and mitigation framework for deep neural networks. In *IJCAI*, volume 2, page 8, 2019.

[31] Jingwei Sun, Ang Li, Louis DiValentin, Amin Hassanzadeh, Yiran Chen, and Hai Li. Fl-wbc: Enhancing robustness against model poisoning attacks in federated learning from a client perspective. *Advances in Neural Information Processing Systems*, 34:12613–12624, 2021.

[32] Hugo Lemarchant, Liangzi Li, Yiming Qian, Yuta Nakashima, and Hajime Nagahara. Inference time evidences of adversarial attacks for forensic on transformers. *arXiv preprint arXiv:2301.13356*, 2023.

[33] Zahra Ghodsi, Mojan Javaheripi, Nojan Sheybani, Xinqiao Zhang, Ke Huang, and Farinaz Koushanfar. zprobe: Zero peek robustness checks for federated learning. *arXiv preprint arXiv:2206.12100*, 2022.

[34] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*, pages 3–18. IEEE, 2017.

[35] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. Ieee, 2017.

[36] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1765–1773, 2017.

[37] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[38] Edward Chou, Florian Tramer, and Giancarlo Pellegrino. Sentinet: Detecting localized universal attacks against deep learning systems. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 48–54. IEEE, 2020.

[39] Yansong Gao, Change Xu, Derui Wang, Shiping Chen, Damith C Ranasinghe, and Surya Nepal. Strip: A defence against trojan attacks on deep neural networks. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 113–125, 2019.

[40] Junfeng Guo, Ang Li, and Cong Liu. Aeva: Black-box backdoor detection using adversarial extreme value analysis. *arXiv preprint arXiv:2110.14880*, 2021.

[41] Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160*, 2016.

[42] Shuanglong Liu, Hongxiang Fan, Xinyu Niu, Ho-cheung Ng, Yang Chu, and Wayne Luk. Optimizing cnn-based segmentation with deeply customized convolutional and deconvolutional architectures on fpga. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2018.

[43] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. Scalehls: a scalable high-level synthesis framework with multi-level transformations and optimizations. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022.

[44] Ahmadreza Azizi, Ibrahim Asadullah Tahmid, Asim Waheed, Neal Mangaokar, Jiameng Pu, Mobin Javed, Chandan K Reddy, and Bimal Viswanath. {T-Miner}: A generative approach to defend against trojan attacks on {DNN-based} text classification. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2255–2272, 2021.

[45] Shuo Wang, Surya Nepal, Carsten Rudolph, Marthie Grobler, Shangyu Chen, and Tianle Chen. Backdoor attacks against transfer learning with pre-trained deep learning models. *IEEE Transactions on Services Computing*, 2020.

[46] Eric Wong, Leslie Rice, and J Zico Kolter. Fast is better than free: Revisiting adversarial training. *arXiv preprint arXiv:2001.03994*, 2020.

[47] M Ivette Gomes and Armelle Guillou. Extreme value theory and statistics of univariate extremes: a review. *International statistical review*, 83(2):263–292, 2015.

[48] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[49] Omkar Parkhi, Andrea Vedaldi, and Andrew Zisserman. Deep face recognition. In *BMVC 2015-Proceedings of the British Machine Vision Conference 2015*. British Machine Vision Association, 2015.

[50] Giorgio Severi, Jim Meyer, Scott E Coull, and Alina Oprea. Explanation-guided backdoor poisoning attacks against malware classifiers. In *USENIX Security Symposium*, pages 1487–1504, 2021.

[51] Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *proceedings of the 25th international conference on world wide web*, pages 507–517, 2016.

[52] Trojai leaderboard. https://pages.nist.gov/trojai/, 01 2021.

[53] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[54] Shih-Lun Chen, Ho-Yin Lee, Chiung-An Chen, Hong-Yi Huang, and Ching-Hsing Luo. Wireless body sensor network with adaptive low-power design for biometrics and healthcare applications. *IEEE Systems Journal*, 3(4):398–409, 2009.

[55] Mohammad Tehranipoor and Cliff Wang. *Introduction to hardware security and trust*. Springer Science & Business Media, 2011.

[56] Brice Colombier and Lilian Bossuet. Survey of hardware protection of design data for integrated circuits and intellectual properties. *IET Computers & Digital Techniques*, 8(6):274–287, 2014.

[57] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE design & test of computers*, 27(1):10–25, 2010.

[58] Swarup Bhunia, Michael S Hsiao, Mainak Banga, and Seetharam Narasimhan. Hardware trojan attacks: Threat analysis and countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, 2014.

[59] Yu Liu, Ke Huang, and Yiorgos Makris. Hardware trojan detection through golden chip-free statistical side-channel fingerprinting. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6, 2014.

[60] Lang Lin, Markus Kasper, Tim Güneysu, Christof Paar, and Wayne Burleson. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 382–395. Springer, 2009.

[61] Rajat Subhra Chakraborty, Francis Wolff, Somnath Paul, Christos Papachristou, and Swarup Bhunia. Mero: A statistical approach for hardware trojan detection. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 396–410. Springer, 2009.

[62] MA Nourian, Mahdi Fazeli, and David Hély. Hardware trojan detection using an advised genetic algorithm based logic testing. *Journal of Electronic Testing*, 34(4):461–470, 2018.

[63] Sayandeep Saha, Rajat Subhra Chakraborty, Srinivasa Shashank Nuthakki, Debdeep Mukhopadhyay, et al. Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 577–596. Springer, 2015.

[64] Mohamed El Massad, Siddharth Garg, and Mahesh V Tripunitara. Integrated circuit (ic) decamouflaging: Reverse engineering camouflaged ics within minutes. In *NDSS*, pages 1–14, 2015.

[65] Yu Liu, Georgios Volanis, Ke Huang, and Yiorgos Makris. Concurrent hardware trojan detection in wireless cryptographic ics. In *2015 IEEE International Test Conference (ITC)*, pages 1–8. IEEE, 2015.

[66] Xiaoxiao Wang, Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. Hardware trojan detection and isolation using current integration and localized current analysis. In *2008 IEEE international symposium on defect and fault tolerance of VLSI systems*, pages 87–95. IEEE, 2008.

[67] Ramesh Karri, Jeyavijayan Rajendran, and Kurt Rosenfeld. Trojan taxonomy. In *Introduction to hardware security and trust*, pages 325–338. Springer, 2012.

[68] Samer Moein, Salman Khan, T Aaron Gulliver, Fayez Gebali, and M Watheq El-Kharashi. An attribute based classification of hardware trojans. In *2015 Tenth International Conference on Computer Engineering & Systems (ICCES)*, pages 351–356. IEEE, 2015.

[69] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. Fanci: identification of stealthy malicious logic using boolean functional analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 697–708, 2013.

[70] Jie Zhang, Feng Yuan, Linxiao Wei, Yannan Liu, and Qiang Xu. Veritrust: Verification for hardware trust. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(7):1148–1161, 2015.

[71] Zhixin Pan and Prabhat Mishra. Automated test generation for hardware trojan detection using reinforcement learning. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 408–413, 2021.

[72] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[73] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3):729, 2012.

[74] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[75] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[76] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. Time limits in reinforcement learning. In *International Conference on Machine Learning*, pages 4045–4054. PMLR, 2018.

[77] Randy Torrance and Dick James. The state-of-the-art in ic reverse engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 363–381. Springer, 2009.

[78] Travis Meade, Shaojie Zhang, and Yier Jin. Netlist reverse engineering for high-level functionality reconstruction. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 655–660. IEEE, 2016.

[79] Wenchao Li, Zach Wasson, and Sanjit A Seshia. Reverse engineering circuits using behavioral pattern mining. In *2012 IEEE international symposium on hardware-oriented security and trust*, pages 83–88. IEEE, 2012.

[80] Marc Fyrbiak, Sebastian Strauß, Christian Kison, Sebastian Wallat, Malte Elson, Nikol Rummel, and Christof Paar. Hardware reverse engineering: Overview and open challenges. In *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, pages 88–94. IEEE, 2017.

[81] Meng Li, Kaveh Shamsi, Travis Meade, Zheng Zhao, Bei Yu, Yier Jin, and David Z Pan. Provably secure camouflaging strategy for ic protection. *IEEE transactions on computer-aided design of integrated circuits and systems*, 38(8):1399–1412, 2017.

[82] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. Camoperturb: Secure ic camouflaging for minterm protection. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2016.

[83] Bicky Shakya, Haoting Shen, Mark Tehranipoor, and Domenic Forte. Covert gates: Protecting integrated circuits with undetectable camouflaging. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 86–118, 2019.

[84] Kaveh Shamsi, David Z Pan, and Yier Jin. On the impossibility of approximation-resilient circuit locking. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 161–170. IEEE, 2019.

[85] Muhammad Yasin, Jeyavijayan JV Rajendran, Ozgur Sinanoglu, and Ramesh Karri. On improving the security of logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(9):1411–1424, 2015.

[86] Muhammad Yasin and Ozgur Sinanoglu. Evolution of logic locking. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6. IEEE, 2017.

[87] Yang Xie and Ankur Srivastava. Anti-sat: Mitigating sat attack on logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(2):199–207, 2018.

[88] Benjamin Tan, Ramesh Karri, Nimisha Limaye, Abhrajit Sengupta, Ozgur Sinanoglu, Md Moshiur Rahman, Swarup Bhunia, Danielle Duvalsaint, Amin Rezaei, Yuanqi Shen, et al. Benchmarking at the frontier of hardware security: Lessons from logic locking. *arXiv preprint arXiv:2006.06806*, 2020.

[89] Bicky Shakya, Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia, and Mark Tehranipoor. Benchmarking of hardware trojans and maliciously affected circuits. *Journal of Hardware and Systems Security*, 1(1):85–102, 2017.

[90] Yipei Yang, Jing Ye, Yuan Cao, Jiliang Zhang, Xiaowei Li, Huawei Li, and Yu Hu. Survey: Hardware trojan detection for netlist. In *2020 IEEE 29th Asian Test Symposium (ATS)*, pages 1–6. IEEE, 2020.

[91] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. A novel technique for improving hardware trojan detection and reducing trojan activation time. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(1):112–125, 2011.

[92] Kan Xiao, Domenic Forte, Yier Jin, Ramesh Karri, Swarup Bhunia, and Mohammad Tehranipoor. Hardware trojans: Lessons learned after one decade of research. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(1):1–23, 2016.

[93] He Li, Qiang Liu, and Jiliang Zhang. A survey of hardware trojan threat and defense. *Integration*, 55:426–437, 2016.

[94] Lawrence H Goldstein and Evelyn L Thigpen. Scoap: Sandia controllability/observability analysis program. In *Proceedings of the 17th Design Automation Conference*, pages 190–196, 1980.

[95] TC Chen, Kapali P Eswaran, Vincent Y Lum, and C Tung. Simplified odd-even sort using multiple shift-register loops. *International Journal of Computer & Information Sciences*, 7(3):295–314, 1978.

[96] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, et al. Triggered instructions: a control paradigm for spatially-programmed architectures. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 142–153. ACM, 2013.

[97] Mark C Hansen, Hakan Yalcin, and John P Hayes. Unveiling the iscas-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, 1999.

[98] Theodore W. Manikas. *MCNC Benchmark Netlists.*, June 28, 2012.

[99] Franc Brglez, David Bryan, and Krzysztof Kozminski. *ISCAS89 Benchmark Netlists.*, September 22, 2006.

[100] Rajat Arora and Michael S Hsiao. Enhancing sat-based bounded model checking using sequential logic implications. In *17th International Conference on VLSI Design. Proceedings.*, pages 784–787. IEEE, 2004.

[101] Zeying Yuan. *Sequential Equivalence Checking of Circuits with Different State Encodings by Pruning Simulation-based Multi-Node Invariants*. PhD thesis, Virginia Tech, 2015.

[102] Seyyed Mohammad Saleh Samimi. Testability measurement tool, 2014.

[103] Ilan Schnell. *pycosat 0.6.3*, Nov 9, 2017.

[104] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. A guide to deep learning in healthcare. *Nature medicine*, 25(1):24, 2019.

[105] The HIPAA Privacy Rule. https://www.hhs.gov/hipaa/for-professionals/privacy/index.html.

[106] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. Cryptodl: Deep neural networks over encrypted data. *arXiv preprint arXiv:1711.05189*, 2017.

[107] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. Low latency privacy preserving inference. In *International Conference on Machine Learning*, pages 812–821. PMLR, 2019.

[108] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Annual International Cryptology Conference*, pages 483–512. Springer, 2018.

[109] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. Faster cryptonets: Leveraging sparsity for real-world encrypted inference. *arXiv preprint arXiv:1811.09953*, 2018.

[110] Amartya Sanyal, Matt Kusner, Adria Gascon, and Varun Kanade. Tapas: Tricks to accelerate (encrypted) prediction as a service. In *International Conference on Machine Learning*, pages 4490–4499. PMLR, 2018.

[111] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Chet: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–156, 2019.

[112] Marshall Ball, Brent Carmer, Tal Malkin, Mike Rosulek, and Nichole Schimanski. Garbled neural networks are practical. *IACR Cryptol. ePrint Arch.*, 2019:338, 2019.

[113] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 619–631, 2017.

[114] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1651–1669, 2018.

[115] Zahra Ghodsi, Akshaj Kumar Veldanda, Brandon Reagen, and Siddharth Garg. Cryptonas: Private inference on a relu budget. *Advances in Neural Information Processing Systems*, 33:16961–16971, 2020.

[116] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[117] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[118] Mohammad Samragh, Siam Hussain, Xinqiao Zhang, Ke Huang, and Farinaz Koushanfar. On the application of binary neural networks in oblivious inference. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4630–4639, 2021.

[119] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[120] Mikhail Atallah, Marina Bykova, Jiangtao Li, Keith Frikken, and Mercan Topkara. Private collaborative forecasting and benchmarking. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 103–114, 2004.

[121] Moni Naor and Benny Pinkas. Computationally secure oblivious transfer. *Journal of Cryptology*, 18(1), 2005.

[122] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Crypto*, volume 2729. Springer, 2003.

[123] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 535–548, 2013.

[124] Sophia Yakoubov. A gentle introduction to yao's garbled circuits, 2017.

[125] Andrew Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, 1986.

[126] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages, and Programming*. Springer, 2008.

[127] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *EuroS&P)*, pages 112–127. IEEE, 2016.

[128] Asit Mishra, Eriko Nurvitadhi, Jeffrey J Cook, and Debbie Marr. Wrpn: Wide reduced-precision networks. *arXiv preprint arXiv:1709.01134*, 2017.

[129] Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar. Rebnet: Residual binarized neural network. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 57–64. IEEE, 2018.

[130] Joseph Bethge, Christian Bartz, Haojin Yang, Ying Chen, and Christoph Meinel. Meliusnet: Can binary neural networks achieve mobilenet-level accuracy? *arXiv preprint arXiv:2001.05936*, 2020.

[131] Zechun Liu, Zhiqiang Shen, Marios Savvides, and Kwang-Ting Cheng. Reactnet: Towards precise binary neural network with generalized activation functions. In *European Conference on Computer Vision*, pages 143–159. Springer, 2020.

[132] Wei Tang, Gang Hua, and Liang Wang. How to train a compact binary neural network with high accuracy? In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[133] Lanlan Liu and Jia Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[134] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. *arXiv preprint arXiv:1812.08928*, 2018.

[135] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 707–721, 2018.

[136] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: programmable, efficient, and scalable secure two-party computation for machine learning. *ePrint Report*, 1109, 2017.

[137] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[138] FaceScrub. *The FaceScrub dataset*, 2020. http://engineering.purdue.edu/~mark/puthesis, (accessed July 3, 2020).

[139] Hong-Wei Ng and Stefan Winkler. A data-driven approach to cleaning large face datasets. In *IEEE international conference on image processing*, 2014.

[140] Malaria Cell Images, accessed on 01/20/2019. https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria.

[141] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, 2016.

[142] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. *arXiv preprint arXiv:1705.08963*, 2017.

[143] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1389–1397, 2017.

[144] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[145] Georgios Damaskinos, Rachid Guerraoui, Rhicheek Patra, Mahsa Taziki, et al. Asynchronous byzantine machine learning (the case of sgd). In *International Conference on Machine Learning*, pages 1145–1154. PMLR, 2018.

[146] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. Analyzing federated learning through an adversarial lens. In *International Conference on Machine Learning*, pages 634–643. PMLR, 2019.

[147] Gilad Baruch, Moran Baruch, and Yoav Goldberg. A little is enough: Circumventing defenses for distributed learning. *Advances in Neural Information Processing Systems*, 32:8635–8645, 2019.

[148] Zahra Ghodsi, Mojan Javaheripi, Nojan Sheybani, Xinqiao Zhang, Ke Huang, and Farinaz Koushanfar. zprobe: Zero peek robustness checks for federated learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4860–4870, 2023.

[149] Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *International Conference on Machine Learning*, pages 5650–5659. PMLR, 2018.

[150] Amine Boussetta, El-Mahdi El-Mhamdi, Rachid Guerraoui, Alexandre Maurer, and Sébastien Rouault. Aksel: Fast byzantine sgd. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*, 2021.

[151] Yudong Chen, Lili Su, and Jiaming Xu. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(2):1–25, 2017.

[152] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[153] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(2):1–19, 2019.

[154] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1074–1091. IEEE, 2021.

[155] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[156] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[157] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[158] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[159] Yann LeCun. The MNIST Database. http://yann.lecun.com/exdb/mnist/, 1998.

[160] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[161] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[162] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). *URL http://www. cs. toronto. edu/kriz/cifar. html*, 5, 2010.

[163] Lukas Burkhalter, Hidde Lycklama, Alexander Viand, Nicolas Küchler, and Anwar Hithnawi. Rofl: Attestable robustness for secure federated learning. *arXiv preprint arXiv:2107.03311*, 2021.

[164] Xiao Wang. EMP-Toolkit. https://github.com/emp-toolkit, accessed 2022.

[165] Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Arsany Hany Abdelmessih Guirguis, and Sébastien Louis Alexandre Rouault. Aggregathor: Byzantine machine learning via robust gradient aggregation. In *The Conference on Systems and Machine Learning (SysML), 2019*, number CONF, 2019.

[166] Sai Praneeth Karimireddy, Lie He, and Martin Jaggi. Byzantine-robust learning on heterogeneous datasets via bucketing. In *International Conference on Learning Representations*, 2022.

[167] Vale Tolpegin, Stacey Truex, Mehmet Emre Gursoy, and Ling Liu. Data poisoning attacks against federated learning systems. In *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I 25*, pages 480–501. Springer, 2020.

[168] Jierui Lin, Min Du, and Jian Liu. Free-riders in federated learning: Attacks and defenses. *arXiv preprint arXiv:1911.12560*, 2019.

[169] Chen Chen et al. Practical attribute reconstruction attack against federated learning. *TBDATA*, 2022.

[170] Camille Brunet-Saumard et al. K-bmom: A robust lloyd-type clustering algorithm based on bootstrap median-of-means. *CSDA*, 2022.