Improving Fault Tolerance of Computing Systems for Autonomous Machines

by

Yiming Gan

Submitted in Partial Fulfillment of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Yuhao Zhu

Department of Computer Science
Arts, Sciences and Engineering
Edmund A. Hajim School of Engineering and Applied Sciences

University of Rochester
Rochester, New York

2023

Dedicated to my loving parents Ning Gan and Yan Zhao, and my fiancée Shiqi Song.

Table of Contents

Biographical Sketch

Yiming Gan was born in Taiyuan, China. He received his Bachelor's Degree in Information Engineering in 2016 from Beijing Institute of Technology. He received his Master's Degree in Computer Engineering in 2018 from University of California, Santa Barbara. He joined the Department of Computer Science at the University of Rochester to become a PhD Student under the supervision of Professor Yuhao Zhu. His research interests lie in computer architecture, with emphasis on building reliable computing systems for autonomous machines with tolerable overhead. During the five years of study as a PhD student, he has published papers on ISCA, MICRO, HPCA, PACT, DAC, ISSRE, etc. He has won the best paper nomination of PACT 2020. He had two internships at ARM and Meta. He organized multiple tutorials and serverd as reviewers for ECCV and IISWC.

He has following publications during his doctoral study:

Conference Articles

- **Yiming Gan**, Yuxian Qiu, Lele Chen, Jingwen Leng, Yuhao Zhu
  Low-Latency Proactive Continuous Vision. Best Paper Nomination.
  In 29th International Conference on Parallel Architectures and Compilation Techniques (PACT'20).
- Igor Fedorov, Marko Stamenovic, Carl Jensen, Li-Chia Yang, Ari Mandell, **Yiming Gan** , Matthew Mattina, Paul N. Whatmough.

TinyLSTMs: Efficient Neural Speech Enhancement for Hearing Aids.
In INTERSPEECH 2020.

- **Yiming Gan**, Yuxian Qiu, Jingwen Leng, Minyi Guo, Yuhao Zhu
  Architecture Support for Robust Deep Learning.
  In 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO'2020).

- **Yiming Gan** , Yu Bo, Boyuan Tian, Leimeng Xu, Wei Hu, Shaoshan Liu, Qiang Liu, Yanjun Zhang, Jie Tang and Yuhao Zhu.
  Eudoxus: Characterizing and Accelerating Localization in Autonomous Machines.
  In 27th IEEE International Symposium on High-Performance Computer Architecture (HPCA-27).

- Weizhuang Liu, Bo Yu, **Yiming Gan**, Qiang Liu, Jie Tang, Shaoshan Liu, Yuhao Zhu.
  Archytas: A Framework for Synthesizing and Dynamically Optimizing Accelerators for Robotic Navigation.
  In 54th IEEE/ACM International Symposium on Microarchitecture (MICRO'2021).

- Yu Feng, Gunnar Hammonds, **Yiming Gan**, Yuhao Zhu.
  Crescent: Taming Memory Irregularities for Accelerating Deep Point Cloud Analytics.
  49th IEEE/ACM International Symposium on Computer Architecture (ISCA '22)

- **Yiming Gan**, Paul Whatmough, Jingwen Leng, Bo Yu, Shaoshan Liu, Yuhao Zhu.
  BRAUM: Analyzing and Protecting Autonomous Machine Software Stack
  33rd IEEE International Symposium on Software Reliability Engineering (ISSRE-22)

- Abhishek Tyagi, **Yiming Gan**, Shaoshan Liu, Bo Yu, Yuhao Zhu.
  Thales: Formulating and Estimating Architectural Vulnerability Factor for DNN Accelerators
  29th IEEE International Symposium on High-Performance Computer Architecture (HPCA-29)

- Yuhui Hao*, **Yiming Gan***, Bo Yu, Qiang Liu, Shaoshan Liu, Yuhao Zhu. (*:co-first author.)
  BLITZCRANK: Factor Graph Accelerator for Motion Planning
  60th Design Automation Conference (DAC)

Journal Articles

- **Yiming Gan**, Yuxian Qiu, Jingwen Leng, Yuhao Zhu
  SVSoC: Speculative Vision Systems-on-a-Chip
  Volume 18, Issue 1, Jan-June 2019, Computer Architecture Letters (CAL)

- Qiang liu, Yuhui Hao, Weizhuang Liu, Bo Yu, **Yiming Gan**, Jie Tang, Shao-Shan Liu, Yuhao Zhu.
  An Energy Efficient and Runtime Reconfigurable Accelerator for Robotic Localization
  IEEE Transcations on Computers

Acknowledgments

First and foremost, I would like to express my appreciation to the Department of Computer Science at the University of Rochester, as well as the city of Rochester itself, for their support throughout this journey. Despite enduring hundreds of snowy and cloudy days over the past five years, I have experienced immense warmth and kindness from the department, the university, and the city. It has been an absolute pleasure to study here, focusing on the research I am passionate about, and living in such a tranquil and beautiful city. With a heavy heart, I bid farewell to this chapter of my life.

I would like to extend my heartfelt gratitude to my advisor, Yuhao Zhu, who has guided me in transforming from a student with limited knowledge, to a budding, independent researcher with a solid foundation. Yuhao has mentored me in every aspect of my academic journey, including reading papers, coding, formulating ideas, collecting evaluation results, writing papers, plotting data, and presenting my work.

Yuhao has profoundly influenced my approach to research and my thought process as a researcher. He has encouraged me to pay attention to details, maintain rigor, and formulate and solve problems logically. Reflecting on my journey, I realize how naive I was when I first joined the department in 2018, and how far I have come as a researcher in my field. It is really a fun fact that now when I teach my students, I say things and try to educate them, and realize that what I said was learned from Yuhao. That's the beauty of education I guess.

I would also like to extend my gratitude to the other members of my committee who have supported me through three six-month reviews and the final defense: Dr. Paul N.

Whatmough, Prof. Chenliang Xu, Prof. Sreepathi Pai, and Prof. Dwarkadas Sandhya. Paul mentored me during my first internship at ARM and provided valuable feedback on my thesis whenever needed. Chenliang introduced me to Advanced Computer Vision, while Sree educated me on Advanced Compiler Techniques. Sandhya taught me Operating Systems, a course in which I admittedly struggled. Ironically, the shortcomings in Operating Systems came back to haunt me during the final project of my thesis, compelling me to delve deeper into the subject for its completion. Additionally, I would like to thank Prof. Jingwen Leng, who provided insightful suggestions on nearly every project I undertook during the past five years.

I am truly grateful for my amazing lab mates, Yu Feng, Tiancheng Xu, and Boyuan Tian. They have been instrumental in supporting my research and coursework. I have shared fantastic moments with Tianlang Chen, Xiong Wei, and Weijiang Li, and cherish the memories of the basketball game morning and our spontaneous 400-mile trips to New York City just to catch a play-off game. I also want to express my appreciation for the encouragement and camaraderie of friends from the old days: Zhaoxiang Hou, Yihan Li, Liang Zhang, Jiacheng Bao, Yichen Wang, Jenny (Yueling) Zeng, and Ling Liang. My university dorm mates have been incredibly supportive, and I am thankful to Hanwen Qiu, Zeyu Wang, Yundong Zhang, Junzhe Zhao, Xingda Zhou, and Bangsheng Zhuo. Although if we played and chated less, I could have graduated earlier.

I was fortunate to have the opportunity to intern at ARM and Meta during the summers. At ARM, I had a fantastic experience working with the Machine Learning group, particularly with Igor Fedorov and Paul. The paper we collaborated on during that internship remains my most cited publication to this day—truly demonstrating the impact of machine learning! I would also like to express my gratitude to Ziyun Li, who mentored me during my internship at Meta. Regrettably, I was unable to pursue further internships at Meta, despite the generous compensation they offered.

Some of my fondest memories in Rochester are the countless Saturday nights spent at Yapeng's apartment, where Lele would bring home sea bass after a day of fishing. I

had a wonderful time with Yapeng Tian, Lele Chen, Jing Shi, Wei Zhu, Chen Yu, and Xinyi Yang during those evenings. Letting go of all exhaustion, we immersed ourselves in delectable food, refreshing beer, and entertaining movies. I can't help but wonder if I will ever experience such sheer joy and camaraderie again in my lifetime.

I am deeply grateful for the funding support I received from the Semiconductor Research Corporation, and I never took this privilege for granted. Unfortunately, due to the COVID-19 situation, I was unable to attend conferences as frequently as I would have liked. These gatherings, where I had the opportunity to learn from outstanding researchers in the industry, were invaluable experiences for me.

I would also like to express my gratitude to my advisors from my undergraduate years at the Beijing Institute of Technology, and during my Master's program at the University of California, Santa Barbara. I am particularly thankful to Prof. Yuan Xie, who introduced me to the field of computer architecture and provided me with the opportunity to work alongside Shuangchen Li, Lei Deng, Xing Hu, and Yu Ji. These individuals have not only become my friends, but also my mentors and teachers.

Most importantly, I want to express my deepest appreciation to my family. My parents, Ning Gan and Yan Zhao, are my biggest fans and supporters. They hold me in such high esteem that I often need to remind them that I still have much to improve. My inherent confidence, optimism, and outlook on life can all be traced back to the values they instilled in me. Moreover, they have consistently provided for me, covering the costs of food, rent, and even rewarding me with 1000 dollars for each first-author paper I publish (co-first authorship also counts). I truly believe that no parents could be more generous and supportive than them. Lastly, I would like to thank my fiancée, Shiqi Song, for always being there for me. Despite spending much of our time in separate countries, I always feel her presence right by my side. She has a naturally pessimistic personality and often worries about whether I will successfully complete my program and obtain my PhD degree. I am overjoyed to tell her that, from now on, she no longer has to worry about that.

Abstract

Autonomous machines, encompassing self-driving vehicles, drones, and mobile robots, are progressively becoming vital components in human society. As the algorithmic complexity employed in these machines expands and the demand for computational resources escalates, ensuring their reliability is more crucial than ever. Traditional approaches to guaranteeing reliability are often uniformly applied to all software, leading to undesirable overheads in system latency, energy consumption, and silicon budget.

In my thesis, I present cross-layer solutions spanning hardware architecture, compilers, operating systems, software, and algorithms to enhance system reliability under various error types while minimizing the resulting overhead. The first part of this thesis works on protecting the perception module from adversarial attacks, in which I propose an efficient adversarial example detector and a dynamic network topology to process standard and adversarial examples dynamically. The second part goes beyond the perception module to protect the entire computing system, in which I propose to utilize the differences in the inherent fault-tolerance levels inside autonomous machine software to selectively apply protection and reduce area overhead. The contributions made in this thesis hold the potential for long-lasting and significant impacts since the aspects explored are pivotal to technological advancement roadmaps. Moreover, the systematic and practical ideas and methodologies outlined in this thesis offer a robust foundation for future research and development in the autonomous machine domain.

Contributors and Funding Sources

This work was supervised by a dissertation committee consisting of Professor Yuhao Zhu, Dr. Paul N. Whatmough, Prof. Chenliang Xu, Prof. Sreepathi Pai and Prof. Dwarkadas Sandhya.

The research presented in Chapter 2 was a collaborative effort with Yuxian Qiu, under the supervision of Professor Jingwen Leng, Professor Minyi Guo and Professor Yuhao Zhu. The work in Chapter 3 was carried out in collaboration with Haotao Wang and Yapeng Tian, supervised by Professor Atlas Wang and Yuhao Zhu. The research covered in Chapters 4 and 5 was conducted under the guidance of Dr. Paul N. Whatmough, Dr. Bo Yu, Dr. Shaoshan Liu, Professor Jingwen Leng, and Professor Yuhao Zhu.

This material is based upon work supported by the Semiconductor Research Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of above named organization.

List of Tables

List of Figures

1 Introduction

Autonomous machines, including self-driving vehicles, unmanned aerial vehicles (UAVs), and mobile robots for transportation and manufacturing, are transforming the way we live and work. By the end of 2025, it is projected that 284 million vehicles will be operating in the United States, with most of them featuring some degree of autonomy (Sudhakar et al., 2022). Similarly, the automated drone market is expected to reach 42.8 billion, with annual sales exceeding 2 million units (Schroth, 2020). Over the past decades, the primary function of these machines has evolved from "following orders" to "creating orders," a trend that has become increasingly pronounced with recent advances in large-scale models (Khan et al., 2022; Han et al., 2022; Dosovitskiy et al., 2020; Radford et al., 2018; Shen et al., 2023). The driving force behind this transformation is the rapid growth of computational capabilities and corresponding innovations in algorithms.

Reliability of computing systems of autonomous machines have gradually become a roadblock. The definition of reliability here is whether computing systems can behave normally when different kinds of potential errors happen. Errors include commonly exist bit flips in hardware (Baumann, 2005a,b; Mukherjee, 2011; Nicolaidis, 2010), adversarial attacks against different algorithms (Madry et al., 2017; Akhtar and Mian, 2018), and potential software bugs (Hangal and Lam, 2002). Such errors, if not properly handled, can cause severe consequences such as system failure and mission failure of

autonomous machines, putting human life in danger.

Throughout the history of computer science, various techniques have been developed to improve fault tolerance. For example, enhancing modular redundancy, both spatially (Anghel et al., 2000) and temporally (Kim, 1999), has been implemented in numerous systems. However, these traditional techniques often struggle to be practical in current autonomous machine applications for three main reasons. First, most autonomous machines operate under real-time constraints (Joseph and Pandya, 1986; Hatley and Pirbhai, 2013), and incorporating reliability-enhancing techniques could violate these constraints, ultimately undermining the usability of the autonomous machines. Second, many autonomous machines feature tight form factors (Rajendran and Smith, 2015), limiting the area and silicon budgets for allocating additional resources to ensure reliability. Third, autonomous machines typically rely on fixed-size batteries for power. Implementing most fault tolerance enhancement techniques leads to increased energy consumption, which ultimately reduces the operating time of autonomous machines. Addressing these challenges requires the development of new techniques that provide enhanced fault tolerance while minimizing the impact on performance, resource utilization, and energy consumption.

Thesis Statement. To build a reliable computing system for autonomous machines, a reliable perception module, along with other modules including localization, planning and control needs to be considered. The perception module is a critical component in autonomous machines, and its safety is often jeopardized by adversarial attacks. To protect the perception module, a perception system that could efficiently detect adversarial examples and a dynamic network topology that could achieve high accuracy on both standard and adversarial images should be built. Apart from perception, other modules such as localization, planning, and control exhibit varying levels of inherent fault tolerance when facing errors. Based on this observation, a selective protection mechanism based on the classification of inherent fault-tolerance levels of different algorithms in autonomous machine software can ensure reliability while not

introducing high performance overhead is needed.

Introduction organization. The remainder of this chapter is structured as follows: In Sec. 1.1, I provide an overview of my contributions to this research area. The potential long-term impact of my work is discussed in Sec. 1.2. The outline of the rest of the dissertation is presented in Sec. 1.3, while Sec. 1.4 enumerates previously published materials used in this dissertation.

1.1 Research Contributions

The primary contribution of my thesis is to enhance the reliability of computing systems in autonomous machines while keeping overhead minimal. I contend that reaching this objective solely through architectural solutions is unfeasible, as ensuring reliability inherently introduces significant overhead. Consequently, the main challenge of my work lies in exploiting the inherent algorithmic characteristics for co-designing hardware and algorithms, ultimately achieving reliability at a reduced cost.

For the perception module, an algorithmic insight is that a "hot-path" type characteristic, similar to the concept in program analysis, can reveal the relationship between a single input and its entire class during neural network inference. This characteristic can be leveraged to detect adversarial examples effectively.

In the entire software stack of autonomous machines, the inherent fault tolerance varies between different modules and algorithms. By exploiting this variation, selectively protecting the more vulnerable nodes can substantially reduce overhead while still maintaining the overall system's reliability.

This dissertation presents four contributions to the cross-stack computing system of autonomous machines, encompassing algorithms, compiler, operating system, and hardware architecture. These contributions are illustrated in Fig. 1.1, with the added enhancements shaded to differentiate them from the existing computing stack.

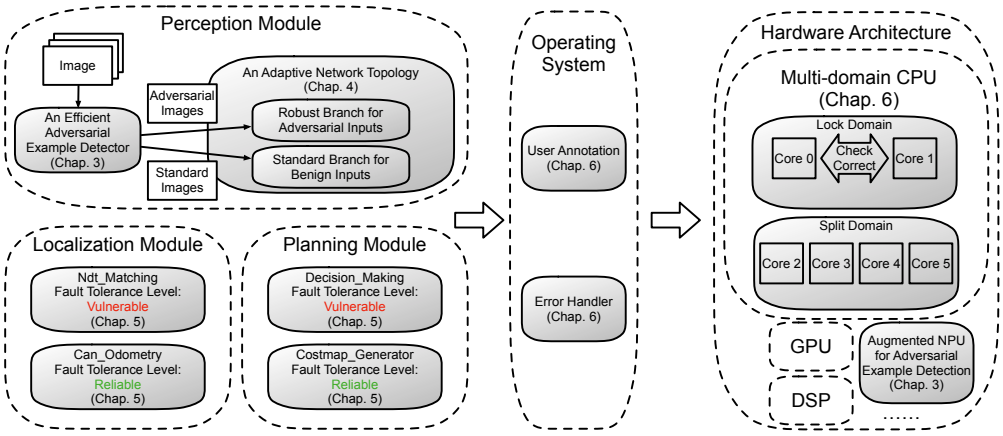


Fig. 1.1: Overview of the contributions in this dissertation.

- I propose PTOLEMY, an algorithm-architecture co-designed system that *detects adversarial attacks at inference time with low overhead and high accuracy*. PTOLEMY exploits the synergies between DNN inference and imperative program execution: an input to a DNN uniquely activates a set of neurons that contribute significantly to the inference output, analogous to the sequence of basic blocks exercised by an input in a conventional program. Critically, I observe that adversarial samples tend to activate distinctive paths from those of benign inputs. Leveraging this insight, I propose an adversarial sample detection framework that uses canary paths generated from offline profiling to detect adversarial samples at runtime. The PTOLEMY compiler, along with the co-designed hardware, enable efficient execution by exploiting the unique algorithmic characteristics.

- I propose MORPHADNET, an adversarial training method that simultaneously improves both standard accuracy (SA) and robust accuracy (RA) over existing methods in a completely automated manner without manual test-time adaptation. The key idea is to condition an adversarial network based on test-time detection of input characteristics. In particular, MORPHADNET proposes to use the prefix layers in a network to predict the attack strength of an input (i.e., perturbation level), which is then used to condition the suffix layers in the network. In essence,

a network trained by our framework can adapt itself depending on the input, avoiding the SA-vs-RA trade-off in existing adversarial training.

- I propose BRAUM, a comprehensive study on the inherent fault tolerance of autonomous machine softwares. By faithfully fault injections, BRAUM is able to study how the AV software stack behaves under different error sources. I show that algorithms in an AV software stack inherently possess different forms of masking mechanisms. Based on the characteristic of the inherent fault tolerance mechanisms, BRAUM formalizes the notion of Fault Tolerance Level (FTL), which quantifies how faults in an algorithm can be masked and/or attenuated without affecting the actuator commands, providing opportunities to relax fault protection.

- I propose KINDRED, a multi-domain lock-step system design that prioritizes high reliability, while minimizing performance overhead. My proposed approach, KINDRED, takes advantage of the inherent diversity in fault tolerance among different tasks in autonomous machine software, scheduling only the vulnerable nodes in the lock-domain. The primary challenge addressed in this work is the intelligent scheduling of tasks across different domains, coupled with efficient error detection and correction in the lock-domain.

1.2 Long-term Impact

The long-term impact of research contributions is crucial for both the domain and society at large. My research contributions will have an enduring influence from two perspectives.

First, the primary issue this dissertation seeks to address is improving system reliability while introducing minimal overhead. The trade-off between reliability and performance will always be a critical aspect of autonomous machine systems. As the

number of autonomous machines increases, the importance of finding a balance between these two factors becomes even more significant.

Second, my research contributions pertain to the field of autonomous machines, a foundational infrastructure in human society. These contributions can be applied to practical use cases of autonomous machines. For instance, most self-driving vehicles currently require more than a 100% increase in chip area for full system duplication. By integrating the findings from BRAUM and KINDRED, the amount of additional silicon required can be significantly reduced, resulting in more efficient and potentially cost-effective solutions for autonomous machines.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 introduces the preliminary knowledge of computing systems for autonomous machines and outlines existing errors that could affect their safety. Chapter 3 presents PTOLEMY, an efficient adversarial example detector for enhancing system reliability. Chapter 4 proposes MORPHADNET, a perception system designed to counter adversarial attacks effectively. Chapter 5 introduces BRAUM, a framework aimed at identifying the fault tolerance classification of different algorithms used in autonomous machines. Chapter 6 utilizes the findings from Chapter 5 to propose a heterogeneous split-lock CPU architecture tailored to the autonomous machine software stack. Chapter 7 offers a retrospective of this dissertation work, summarizing the principles behind building a reliable computing system for autonomous machines and discussing potential future research directions. Chapter 8 concludes the dissertation, highlighting the key contributions and their implications for the field of autonomous machines.

1.4 Previous Published Material

This dissertation contains materials that are previously published in peer-reviewed conferences and journals:

- Chapter 3 contains results from *Ptolemy: Architecture Support for Robust Deep Learning*. Yiming Gan * , Yuxian Qiu *, Jingwen Leng, Minyi Guo, and Yuhao Zhu. In 53rd IEEE/ACM International Symposium on Microarchitecture.
- Chapter 6 contains results from *BRAUM: Analyzing and Protecting Autonomous Machine Software Stack*. Yiming Gan, Paul Whatmough, Jingwen Leng, Bo Yu, Shaoshan Liu, Yuhao Zhu. In 33rd IEEE International Symposium on Software Reliability Engineering.

2 Background: Autonomous Machines and Their Reliabilities

In this chapter, I provide an overview of the background information related to this work. First, I briefly introduce the computing stack, encompassing both hardware and software components, utilized in autonomous machines (Chapter 2.1). Next, I outline the potential reliability concerns within the computing system (Chapter 2.2). Together, these sections demonstrate that the computing system of autonomous machines is complex and operates under potentially threatening environments. I further give definitions of some technical concepts I use in this thesis to avoid confusions. (Chapter 2.3).

2.1 Computing Stack in Autonomous Machines

Autonomous machines have increasingly become a crucial infrastructure in human society. With the rapid growth in the number of vehicles with self-driving capabilities, drones, and mobile robots, autonomy in machines is gaining prominence. There are three essential components to enable autonomy in machines (Vamvoudakis et al., 2015): A sensor system, which includes cameras (Salman et al., 2017; Lee et al., 2013), LiDARs (Verucchi et al., 2020; Hecht, 2018; Ackerman, 2016), radars (Hakobyan and Yang, 2019), and other sensors, captures reflections and changes in the environment. A computing system collects the data from the sensor system and calculates the au-

tonomous machine's behavior, such as its mission and motion. A mechanical system implements the results generated by the computing system, enabling the machine to interact with its environment. My dissertation primarily focuses on the computing system and its reliability, as this plays a critical role in ensuring the safe, effective operation of autonomous machines.

The computing stack of autonomous machines, analogous to the human brain, is extremely complex. The computing stack can vary significantly depending on the specific scenario or application in which the autonomous machine is operating. For instance, the computing stack of a self-driving vehicle (Liu et al., 2020) is considerably more intricate compared to that of a mobile robot (Kortenkamp et al., 1998; Rubio et al., 2019). This complexity arises due to the diverse requirements, sensors, and operations involved in differing autonomous machines' contexts.

Software. I provide an overview of a general autonomous machine system, using a self-driving vehicle as an example, which consists of five key components: sensing, perception, localization, planning, and control. Sensor samples are first synchronized and processed before being used by the perception and localization modules. Perception module attempts to understand the surrounding environment by detecting and tracking objects using the processed sensor data. The localization module positions the vehicle within a global map, leveraging information from the sensors. The planning module utilizes the perception and localization results to plan a path and generate control commands. The control module smooths the control signals and transmits them through the CAN bus to the vehicle's Engine Control Unit (ECU). The control signals then control the vehicle's actuators, such as the gas pedal, brake, and steering wheel. Each module may contain one or more nodes or kernels. A kernel represents an individual process that runs within the system.

Sensing. The beginning of the autonomous machine software is the sensing module. After the sensor system captures signals from the environment, the sensing module of the computing system typically has two tasks. First, it handles sensor data prepro-

cessing. For instance, various image preprocessing methods (Bose and Meyer, 2003; Nakamura, 2017) are applied, including denoising (Motwani et al., 2004; Muresan and Parks, 2003), demosaicing (Li et al., 2008; Hirakawa and Parks, 2006), and sharpening (Dian et al., 2018), to the raw pixels captured by cameras in order to produce high-quality images for perception and localization modules. For LiDAR signals, typical preprocessing procedures involve LiDAR point downsampling and filtering (Meng et al., 2010; Montealegre et al., 2015), which help refine the data for further processing by other modules in the system.

The second task of the sensing module is sensor data synchronization (Faizullin et al., 2022; Yuan et al., 2022), which is crucial for perception and localization algorithms, as they usually perform multi-sensor fusion. For example, many autonomous vehicles use Visual-Inertial Odometry (VIO), which combines data from both camera and IMU sensors. Sensor synchronization ensures that two sensor samples capturing the same event have the same timestamp, allowing for accurate integration of information from multiple sources.

Localization. Localization is a fundamental module to autonomous machines. It calculates the six degrees of freedom (DoF) pose to locate the position and orientation of an agent itself. The six DoF includes the three DoF for the translational pose, which specifies the $< x, y, z >$ position, and the three DoF for the rotational pose, which specifies the orientation about three perpendicular axes, i.e., yaw, roll, and pitch.

Localization algorithms usually have high computational requirements. These algorithms, such as Simultaneous Localization and Mapping (SLAM) (Taketomi et al., 2017; Mur-Artal et al., 2015) and Visual Inertial Odometry (VIO) (Bloesch et al., 2015), perform two main tasks. First, they capture correspondence in continuous frames. Then, they use multiple correspondences to solve a complex optimization problem for the final pose. A typical SLAM algorithm takes tens of milliseconds (ms) latency even when running on a powerful Intel CPU (Liu et al., 2019b; Yoon and Raychowdhury, 2020). VIO algorithms exhibit similar latencies (Suleiman et al., 2018).

Perception. The perception module usually consists of multiple nodes, such as object detection, object tracking, prediction, and multi-sensor fusion (Rosique et al., 2019; Gupta et al., 2021). The perception module first detects and tracks objects from frames captured by cameras or LiDAR points. By combining the historical trajectory and current velocity, the agent predicts the future trajectory of objects within its field of view. Autonomous machines with small form factors may use only one sensor for perception. In contrast, most large-scale autonomous machines, such as self-driving cars, fuse prediction results from multiple sensors to generate an area map. This map is typically represented as a grid map, with zero values representing free space and one values indicating space that is going to be occupied by other objects. The generated map then helps the autonomous system navigate and interact with its environment more effectively.

The perception module is inherently computationally intensive and typically contributes the longest latency in autonomous machines. The serial processing of detection, tracking, and prediction in the perception pipeline exacerbates the computing latency. Although most perception nodes have been accelerated by GPUs or other specialized hardware accelerators, the perception stage still takes more than 100 milliseconds to complete.

Planning. Planning aims to find a collision-free path from the current location to the destination (Fan et al., 2018; Sudhakar et al., 2020). Planning algorithms typically rely on the occupancy grid generated by the perception module, which indicates whether locations are free or occupied. Once the occupancy grid map is created, it remains unchanged during a single planning process. The starting location on the occupancy grid map is determined by the localization module, which takes the current position and orientation of the autonomous machine into account.

Decision-Making. Autonomous machines use state machines to control behavior, with the state machines being controlled by the decision-making module. For example, when an autonomous vehicle detects a pedestrian nearby, the decision-making module

changes the vehicle's status from driving to stopping. Decision-making also relies on the results of perception and localization, as it takes information about the environment and the machine's position into account to make appropriate decisions.

Control. The control module is the last component in the computing stack of autonomous machines and is responsible for smoothing the control commands. It will generate outputs such as the commands on steering wheels, gas pedals to the mechanical system and control the vehicle.

Hardware. Autonomous machine software is complex, and many modules are computationally intensive. To run this software, most autonomous machines are equipped with powerful System-on-a-Chip (SoC) solutions (Ditty, 2022). These SoCs typically contain multiple accelerators such as GPUs, DSPs, and NPUs to execute various tasks in parallel, working together to perform the diverse and demanding tasks required for autonomy. However, at the same time, the increased complexity of these chips with multiple silicon components can lead to higher error rates. This trade-off highlights the importance of balancing performance and reliability to ensure the overall stability and safety of autonomous machines.

2.2 Potential Errors in Computing Systems of Autonomous Machines

Autonomous machines operate in environments with numerous potential threats, and their computing systems can be subject to errors at both the hardware and software levels. These errors, when they occur in the computing system, can significantly affect the system's outputs, ultimately impacting the behavior and safety of autonomous machines. In this section, I will introduce some common errors that may arise in the computing system of autonomous machines.

Soft Errors. In this dissertation, I consider hardware bit-flips that occur within a

single compute cycle. This type of fault is commonly referred to as a soft error and is among the most dominant errors affecting autonomous machine systems. The increasing impact of soft errors has been recently emphasized by industrial studies (Hochschild et al., 2021), where factors such as radiation and temperature changes can result in random bit flips in silicon flip-flop units and memory cells.

Soft errors can lead to various misbehaviors in autonomous machine systems. One of the most common consequences of soft errors is silent data corruption (SDC), where the results deviate from the ground truth value without any notification or indication of error (Dixit et al., 2021). In other cases, soft errors can cause processes to hang or crash. In the context of autonomous machines, hangs and crashes due to soft errors can typically be mitigated by directly restarting the affected process. However, SDCs can potentially cause more severe errors within the system. For instance, an SDC affecting a control command can change an intended decelerating command into an unexpected accelerating command, posing a significant safety risk.

Adversarial Attacks. Unlike soft errors, which are random hardware-level errors, adversarial attacks refer to intentionally manipulated inputs designed to compromise the correctness of targeted algorithms. Many algorithms in the perception module of autonomous machines utilize different forms of deep neural networks (Wen and Jo, 2022; Jebamikyous and Kashef, 2022; Gupta et al., 2021). However, deep neural networks have been proven to be vulnerable to adversarial attacks (Madry et al., 2017; He et al., 2019; Huang et al., 2017; Chakraborty et al., 2018; Guo et al., 2019; Liu et al., 2016; Papernot et al., 2016a). For example, slight changes in pixel values of an input image or placing a sticker on a real-world object (Li et al., 2019) can cause an object detector to misclassify an object, potentially leading to a stop sign being misidentified as a traffic light or something else entirely.

After initially being applied to perception modules, adversarial attacks have been further developed to target other modules of autonomous machines. Researchers have successfully demonstrated adversarial attacks on localization modules (Yang and

Huang, 2019; Patil et al., 2021) and planning modules (Vemprala and Kapoor, 2021; Wang and Gursoy, 2023). These attacks can degrade the performance and reliability of various computing components in autonomous systems.

Software Bugs. Software bugs are another source of potential errors that can impact autonomous machine systems. Similar to traditional software development, autonomous machine software can suffer from various types of bugs, including logic errors, memory errors, and boundary errors (Garcia et al., 2020; Taylor et al., 2021). These software bugs can compromise the correctness of the computing systems used in autonomous machines, leading to incorrect outputs and, potentially, unsafe behavior.

2.3 Technical Concepts

To minimize ambiguity, I will first provide a concise introduction to several key technical concepts utilized throughout this dissertation. Please note that these explanations are not formal definitions, but rather my own interpretations of these concepts, tailored for the context of this research.

Error, fault and failure. Throughout this dissertation, I use the terms error, fault, and failure interchangeably to represent a single concept. This encompasses any unexpected behavior occurring at the hardware or software level, resulting in mistakes. Such unexpected behavior can range from a single bit flip within the hardware to traditional software bugs, as described in Chapter 2.2.

Reliability, robustness and resilience. In this dissertation, the terms reliability, robustness, and resilience collectively denote the ability of a computing system to maintain functionality and produce accurate results within a faulty environment.

Safety and usability. In this dissertation, safety and reliability are considered distinct concepts. When a computing system generates incorrect output due to an error, the impact on safety or usability may vary – in some cases, they might remain unaffected, while in other instances, they could be compromised.

3 An Efficient Adversarial Example Detector

3.1 Introduction

Deep Neural Networks (DNN) are not robust. Small perturbations to inputs could easily "fool" DNNs to produce incorrect results. By manipulating the perturbation, a range of so-called *adversarial attacks* have been demonstrated to lead DNNs to mis-predict (Papernot et al., 2016c; Carlini and Wagner, 2017b; Kurakin et al., 2016; Nguyen et al., 2015; Hu et al., 2020b), which could result in potentially severe consequences. For instance, physically putting a sticker on a stop sign could lead a well-trained object recognition DNN to misclassify the stop sign as a yield sign (Kurakin et al., 2016). Beyond mission-critical scenarios such as autonomous driving, the robustness issue also obstructs the deployment of DNN in privacy/security-sensitive domains such as biometric authentication (Parkhi et al., 2015; Sun et al., 2015).

We take a first step toward architectural support for robust deep learning. For a robustness scheme to be effective in practice, it not only has to accurately *detect* adversarial inputs, but must also do so efficiently *at inference time* so that proper measure could be taken. This paper propose PTOLEMY, an algorithm-architecture co-design system that *detects adversarial attacks at inference time with low overhead and high accuracy*. This enables applications to reject incorrect results produced by adversarial

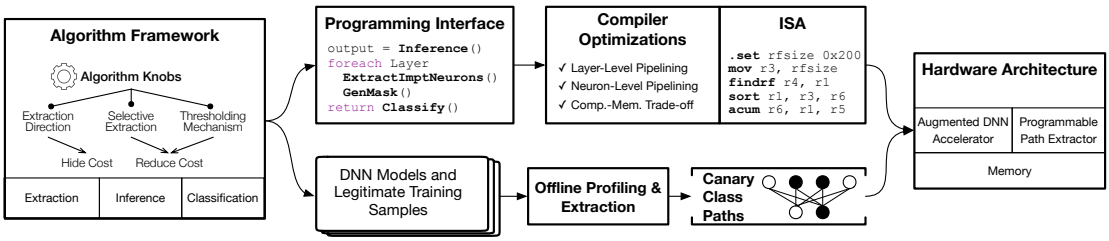


Fig. 3.1: PTOLEMY system overview.

attacks during inference. Fig. 3.1 provides an overview of the system.

Existing countermeasures to adversarial attacks are unable to detect adversarial samples at inference time (Carlini and Wagner, 2017a; He et al., 2017). Fundamentally, they treat DNN inferences as black boxes, ignoring their runtime behaviors. To enable efficient online adversarial detection, this paper takes a different, white-box approach. We exploit the fact that each input to a DNN uniquely exercises an *activation path*—a collection of neurons that contribute significantly to the inference output, analogous to the sequence of basic blocks exercised by an input in a conventional program. Analyzing "hot" activation paths in DNNs, our **key observation** is that inputs that lead to the same inference class tend to exercise a group of paths that are distinctive from other inference classes.

We propose a general algorithmic framework that exploits the runtime path behaviors for efficient online adversarial sample detection. The detection framework constructs a canary *class path* offline for each inference class by profiling the training data. At runtime, it builds the activation path for an input, and detects the input as an adversary if the activation path is different from the canary path associated with the predicted class. The general algorithm framework exposes a myriad of design knobs affecting the critical trade-off between detection accuracy and compute cost, such as how a path is formulated and when the path is constructed. To widen the applicability of our detection framework, PTOLEMY provides a high-level programming interface, which allows programmers to calibrate the algorithmic knobs to explore the accuracy-cost trade-off that best suits an application's needs.

PTOLEMY provides an efficient execution substrate. The key to the execution efficiency is the PTOLEMY compiler, which hides and reduces the detection overhead by exploiting the unique parallelisms and redundancies exposed by the detection algorithms. We show that with the aggressive compile-time optimizations and a well-defined ISA, detection algorithms can be implemented on top of existing DNN accelerators with a set of basic, yet principled, hardware extensions, further widening the applicability of PTOLEMY.

PTOLEMY enables highly accurate adversarial detection with low performance overhead. Compared to today's defense mechanisms that introduce over 10 $\times$ performance overhead, we demonstrate a system that achieves higher accuracy with only 2% performance overhead. PTOLEMY defends not only existing attacks, but also *adaptive* attacks that are specifically designed to defeat our defense (Carlini et al., 2019). We also demonstrate the PTOLEMY framework's flexibility by presenting a range of algorithm variants that offer different accuracy-efficiency trade-offs. For instance, PTOLEMY could trade 10% performance overhead for 0.03 higher detection accuracy.

The PTOLEMY artifact, including the pre-trained models, offline-generated class paths, code to generate adaptive and non-adaptive attacks, and the detection implementation is available at `https://github.com/Ptolemy-DL/Ptolemy`. In summary, PTOLEMY provides a generic framework for low-overhead, high-accuracy online defense against adversarial attacks with the following contributions. First, We propose a novel static-dynamic collaborative approach for adversarial detection by exploiting the unique program execution characteristics of DNN inferences that are largely ignored before. Second, We present a general algorithmic framework, along with a high-level programming interface, that allows programmers to explore key algorithm design knobs to navigate the accuracy-efficiency trade-off space. Third, We demonstrate that with a carefully-designed ISA, compiler optimizations could enable efficient detection by exploiting the unique parallelisms and redundancies exposed by our detection algorithm framework. Finally, We present a programmable hardware to achieve low-latency

online adversarial defense with principled extensions to existing DNN accelerators.

3.2 Background

Adversarial Attacks DNNs are not robust to adversarial attacks, where DNNs mis-predict under slightly perturbed inputs (Carlini and Wagner, 2017b; Kurakin et al., 2016; Papernot et al., 2016c; Moosavi-Dezfooli et al., 2016a). Fig. 3.2 shows one such example, where the two slightly different images are both perceived as stop signs to human eyes, but the second image is mis-predicted by a DNN model as a yield sign. The perturbations could be the result of carefully engineered attacks, but could also be an artifact of normal data acquisition such as noisy sensor capturing and image compression/resizing (Thang and Matsui, 2019).

Fig. 3.2: Adversarial example using the FGSM (Goodfellow et al., 2014a) attack.

Formally, given a DNN $\mathcal{C}$, an input x' is defined as an adversarial sample if it is close to x yet makes $\mathcal{C}^*(x) = \mathcal{C}(x) \neq \mathcal{C}(x')$, where $\mathcal{C}^*(x)$ is the correct class of x. Different adversarial samples differ in their measures of the distance between x and x'. The distance could be small, where the input perturbations are imperceptible to humans (as in the example above), but could also be large, where the perturbations are visible to humans but still "fool" a DNN. For instance, physically putting a sticker on a stop sign could mislead a DNN to misclassify the stop sign as a yield sign (Kurakin et al.,

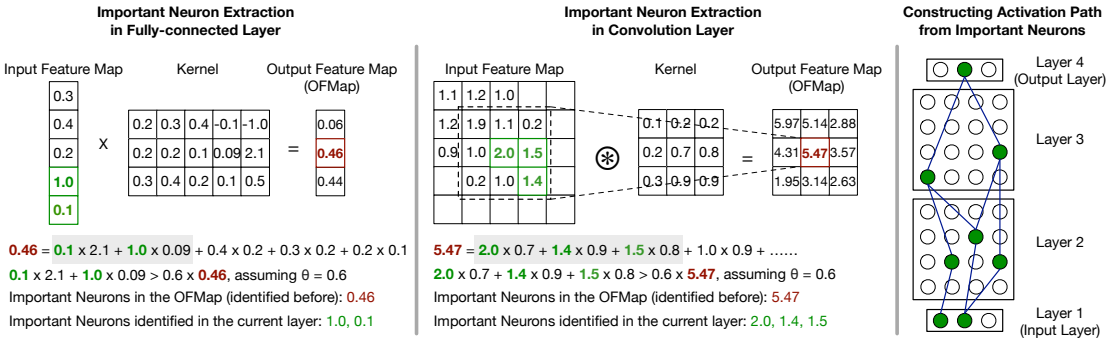


Fig. 3.3: Extracting important neurons from a fully connected layer (left) and a convolution layer (middle), and constructing the activation path from important neurons across layers (right). Activation paths are input-specific. This figure illustrates backward extraction using a cumulative threshold. Forward extraction would start from the first layer rather than from the last year. Absolute thresholding would select important neurons based on absolute partial sums rather than cumulative partial sums.

2016; Li et al., 2019). PTOLEMY targets the general robustness issue that introduces mis-predictions through input perturbations—small or large, inadvertent or malicious. For simplicity, we refer to all of them as adversarial attacks throughout this paper.

An adversarial attack is a black-box attack if it does not assume knowledge of the attacked model; white-box attacks in contrast assume full knowledge of the model. Orthogonally, adaptive attacks have complete knowledge of the defense's inner workings, i.e., are specifically designed to attempt to defeat a defense, while non-adaptive attacks do not (Tramer et al., 2020; Carlini et al., 2019; Carlini and Wagner, 2017a). We show that our detection scheme can defend against a range of different attacks, including the strongest form of attack that can be injected at inference phase (Goldwasser et al., 2022): white-box adaptive attacks.

Countermeasures We aim to enable fast and accurate systems that can *detect* adversarial examples at *inference-time* such that proper measures could be taken. Today's defense mechanisms largely fall under two categories, neither of which meets this goal. The first class of defenses improves the robustness of DNN models at *training time*

(e.g., adversarial retraining) (Tramèr et al., 2017a; Zheng et al., 2016) by incorporating adversarial examples into the training data. However, re-training is not suitable at inference-time and requires accesses to the training data. Another class of defenses uses redundancies to defend against adversarial attacks (Thang and Matsui, 2019; Rouhani et al., 2018), similar to the multi-module redundancy used in classic fault-tolerant systems (Sorin, 2009). This scheme, however, introduces high overhead, limiting its applicability at inference time.

3.3 Algorithmic Framework

This section introduces the PTOLEMY algorithm framework, which enables adversarial attack detection at inference-time with high accuracy and low latency. PTOLEMY provides a set of principled design knobs to allow programmers to customize the accuracy vs. efficiency trade-off.

We first describe the intuition and key concepts behind our algorithm framework (Chapter 3.3.1). We then introduce the algorithm framework, and show that a basic algorithm under the framework introduces excessive compute and memory cost (Chapter 3.3.2). We further introduce key algorithmic knobs that enable different algorithm variants to offer different accuracy-efficiency trade-offs (Chapter 3.3.3). Finally, we introduce a high-level programming interface to flexibly express detection algorithms within our framework (Chapter 3.3.4).

3.3.1 Intuition and Key Concepts

Intuition Each input to a DNN activates a sequence of neurons. We find that inputs that are correctly predicted as the same class tend to activate a unique set of neurons distinctive from that of other inputs. This is a manifestation of recent work on *class-level* model sparsity (Qiu et al., 2019; Wang et al., 2018a), which shows that a small,

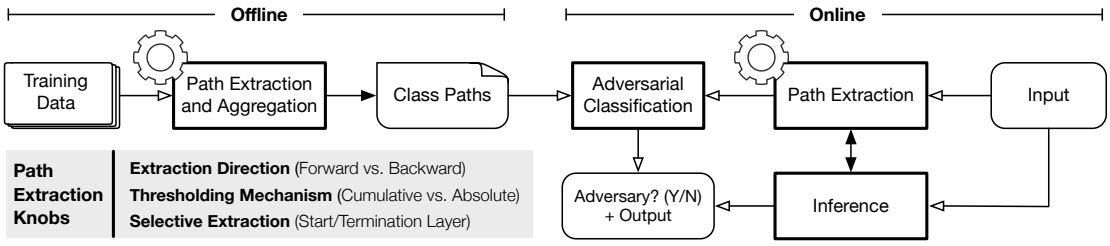
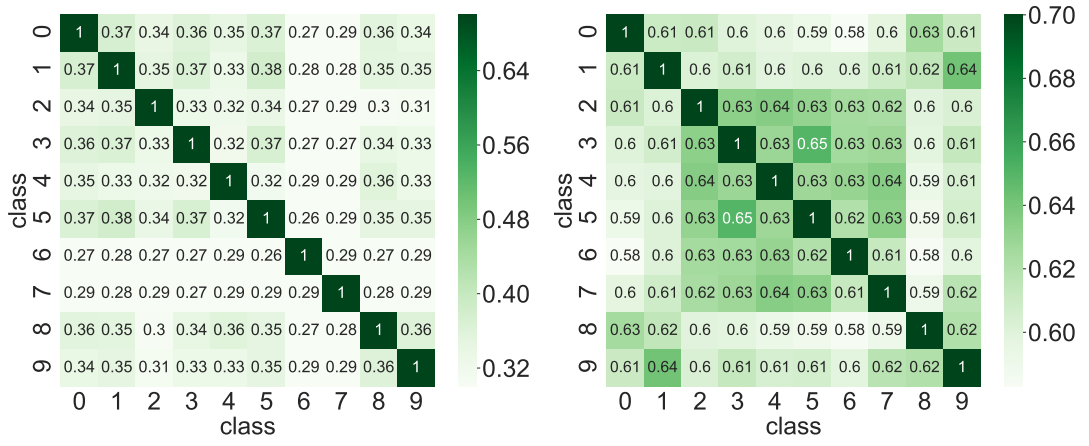


Fig. 3.4: Adversarial detection algorithm framework. It provides a range of knobs for path extraction, which dominates the runtime overhead. Note that the path extraction methods in both the offline and online phases must match.

but distinctive, portion of the network contributes to each predicted class. Taking this perspective, the way adversarial samples alter the inference result can be thought of as activating a sequence of neurons different from the canonical sequence associated with its predicted output. Analyzing dynamic paths in DNN inferences thus allows us to detect adversaries.

A sequence of activated neurons is analogous to a sequence of basic blocks exercised by an input to a conventional program. The frequently exercised basic block sequences, i.e., "hot paths" (Ball and Larus, 1996; Fisher, 1981; Chang and Hwu, 1988), can be used to improve performance in classic profile-guided optimizations and dynamic compilers (Smith, 2000; Smith and Nair, 2005; Donovan et al., 2000). Our approach shares a similar idea, where we treat a DNN as an imperative program, and leverage its runtime paths (sequence of neurons) to guide adversarial sample detection. Conventional countermeasures largely ignore the program execution behaviors of DNN inferences.

Important Neurons The premise of our detection algorithm framework is the notion of *important neurons*, which denote a set of neurons that contribute significantly to the inference output. Important neurons are extracted in a backward fashion. The last layer L_n has only one important neuron, which is the neuron $\mathbf{n}$ that corresponds to the predicted class. At the second last layer L_{n-1}, the important neurons are the minimal set of neurons in the input feature map that contribute to at least θ ($0 \leq \theta \leq 1$) of $\mathbf{n}$.



(a) AlexNet @ ImageNet.

(b) ResNet18 @ CIFAR-10.

Fig. 3.5: Class path similarity ($\theta = 0.5$).

Here, θ controls the coverage of important neurons. To extract the important neurons of layer L_{n-1}, we simply rank the partial sums used to calculate $\mathbf{n}$, and choose the minimal number of neurons whose partial sums collectively contribute to at least $\theta \times \mathbf{n}$.

The left panel in Fig. 3.3 shows an example using a fully-connected layer. Assuming $\theta = 0.6$ and the second neuron in the output feature map (0.46) is the important neuron identified in the next layer. The fourth (1.0) and the fifth (0.1) neurons in the input feature map are identified as the important neurons in the current layer, because they contribute the two large partial sums and their cumulative partial sum (0.3) contribute to more than 60% of the important neuron in the output feature map. The same process can be extended to convolution layers. The middle panel in Fig. 3.3 shows an example. For the important neuron in the output feature map, we first find its receptive field in the input feature map, and then identify the minimal set of neurons in the receptive field whose cumulative partial sums contribute to at least $\theta \times \mathbf{n}$.

This process is repeated *backwards* from the last layer to the first layer, as shown in the right panel in Fig. 3.3. The important neurons identified at layer L_i are used to determine the important neurons at layer L_{i-1}.

From Neurons to Paths The collection of important neurons across all the layers under a given input constitutes an *activation path* of that input, similar to how a sequence of basic blocks constitutes a path/trace in a program. We represent a path using a bitmask, where each bit $m_{i,j}$ indicates whether the neuron (input feature map element) at layer i position j is an important neuron.

From individual activation paths, we introduce the concept of a *class path* for a class c, which aggregates (bitwise OR) the activation paths of different inputs that are correctly predicted as class c. That is: $\mathrm{P}_c = \bigcup_{x \in \bar{x}_c} \mathrm{P}(x)$, where $\mathrm{P}(x)$ denotes the activation path of input x, $\bar{x}_c$ denotes the set of all the correctly predicted inputs of class c, $\bigcup$ denotes bitwise OR, and P_c denotes the class path of class c. We observe that P_c starts to saturate around 100 images and including more images from the training dataset does not result all bits being 1. We do not manually stop filling the bits.

Critically, class paths are significantly different from each other. Fig. 3.5a shows the path similarity in AlexNet (Krizhevsky et al., 2012) across 10 randomly-sampled classes from ImageNet (Deng et al., 2009). Fig. 3.5b shows the path similarity in ResNet18 (Szegedy et al., 2017a) across the 10 classes in CIFAR-10 (Krizhevsky et al., 2009). All the results are obtained on the training set. The average inter-class path similarity is only 36.2% (max 38.2%, 90-percentile 36.6%) for AlexNet on ImageNet and 61.2% (max 65.1%, 90-percentile 63.4%) for ResNet18 on CIFAR-10, suggesting that class paths are distinctive. In an attempt to normalize the dataset, we also perform the same experiment on ResNet50 on ImageNet. The average inter-class path similarity is 37.6% (max 40.9%, 90-percentile 39.1%), similar to those of AlexNet on ImageNet.

The class path similarity is much higher in CIFAR-10 than in ImageNet. This is because ImageNet has 1,000 classes that cover a wide range of objects and CIFAR-10 has only 10 classes, which are similar to each other (e.g., cat vs. dog). The randomly picked 10 classes in ImageNet are more likely to be different from each other than the 10 classes in CIFAR-10. Across all the 1,000 classes in ImageNet, the maximum inter-class path similarity is still only 0.44, suggesting that our random sampling of

ImageNet is representative.

3.3.2 Detection Framework and Cost Analysis

We leverage the clear distinction across different class paths to detect adversarial inputs. If an input x is predicted as class c while its activation path $\mathrm{P}(x)$ does not resemble the class path P_c, we hypothesize that the input is an adversary.

Framework Fig. 3.4 shows an overview of the algorithm framework, which requires static-dynamic collaboration. The static component profiles the training data to extract activation paths $\mathrm{P}(x)$ for each correctly predicted sample x, and generates the class path P_c for each class c as described before. The class paths are stored offline and reused over time. Critically, our profiling method can easily integrate new training samples, whose activation paths would simply be aggregated (`OR`-ed) with the existing class paths without having to re-generate the entire class paths from scratch.

At inference-time, the dynamic component extracts the path for a given input. Note that activation paths are extracted only after the entire DNN inference finishes, because the identification of important neurons starts from the predicted class in the last layer and propagates backward. We will show other variants in Chapter 3.3.3 that relax this restriction.

Given the activation path $\mathrm{P}(x)$ of an input x and the canary class path P_c, where c is the predicted class of x, a classification module then decides whether x is an adversary or not based on the similarity between $\mathrm{P}(x)$ and P_c. While a range of similarity metrics and algorithms could be used, we propose a lightweight algorithm that is extremely efficient to compute while providing high accuracy. Specifically, we first estimate the similarity S between $\mathrm{P}(x)$ and P_c: $\mathrm{S} = \|\mathrm{P}(x)\ \&\ \mathrm{P}_c\|_1 / \|\mathrm{P}(x)\|_1$, where $\|P\|_1$ denotes the number of 1s in the vector P, and $\&$ denotes bitwise `AND`. S is fed into a learned classifier, for which we use the lightweight random forest method (Liaw et al., 2002), for the final classification. The classification module is lightweight, contributing to less

than 0.1% of the total detection cost.

Cost Analysis The algorithm described above is able to achieve accuracy higher than state-of-the-art methods (see Chapter 6.8). However, runtime extraction of activation paths also introduces significant memory and compute costs.

The memory cost is significant because every single partial sum generated during inference must be stored in the memory before the path extraction process begins. The detection algorithm introduces 9 $\times$ to 420 $\times$ memory overhead, which is a lower bound of the actual memory traffic overhead in real systems because the massive partial sums will not be buffered completely on-chip. Storing partial sums will also stall the computing units and increase latency.

Path extraction also introduces compute overhead due to sorting and accumulating partial sums. Using AlexNet as an example, at $\theta = 0.9$, the compute overhead could be as high as 30%. At first glance, it might be surprising that the compute overhead is "only" 30%. Further investigations show that percentage of important neurons in a network is generally below 5% even with $\theta = 0.9$. Thus, the expensive sorting and accumulation operations are applied to only a small portion of partial sums. Note that the compute cost shown here leads to much higher latency overhead in reality because, while inference is massively parallel, sorting and accumulating are much less so. A pure software implementation of the detection algorithm introduces 15.4$\times$ and 50.7$\times$ overhead over inference on AlexNet and ResNet50, respectively.

3.3.3 Algorithmic Knobs and Variants

To trade little accuracy loss for significant efficiency gains, we introduce three algorithmic knobs that control how activation paths are extracted, which dominates the runtime performance/energy overhead. The result is a set of algorithm variants that follow the same algorithm framework described in Fig. 3.4, but that differ in how the paths are extracted.

✿ Hiding Detection Cost: Extraction Direction

The cost introduced by the basic detection algorithm directly increases the inference latency because path extraction and inference must be serialized. We identify a key algorithmic knob that provides the opportunity to hide the compute cost of detection by overlapping detection with inference.

The key to the new algorithm is to extract important neurons in a *forward* rather than a backward manner. Recall that in the original backward extraction process, we use the important neurons in layer L_i's output (which is equivalent to layer L_{i+1}'s input) to identify the important neurons in layer L_i's input. In our new forward extraction process, as soon as layer L_i finishes inference we first determine the important neurons in its output by simply ranking output neurons according to their numerical values and selecting the largest neurons, instead of waiting until after the extraction of layer L_{i+1}. In this way, the extraction of important neurons at layer L_i and the inference of layer L_{i+1} can be overlapped.

✿ Reducing Detection Cost: Thresholding Mechanism

The forward extraction process hides the extraction behind inference, but does not reduce the detection cost, which could significantly increase the energy overhead.

To reduce the detection cost, we propose to extract important neurons using absolute thresholds rather than cumulative thresholds. Whenever a partial sum is generated during inference it is compared against an absolute threshold ϕ. A single-bit mask is stored to the memory based on the comparison result. Later during path extraction, the masks (as opposed to partial sums) are loaded to determine important neurons. Thresholding can be specified at each layer, and can be applied to both extraction directions.

Using absolute thresholds significantly reduces both the compute and memory costs (Chapter 3.7.3), because comparing partial sums against a threshold is much cheaper than sorting and accumulating them, and writing single-bit masks rather than partial sums significantly reduces the memory accesses.

Table 3.1: Summary of PTOLEMY instructions. Operands in the first three instruction classes are registers to simplify encoding.

Class	Name	23-20	19-16	15-12	11-8	7-4	3-0
Inference	`inf`	0000	Input addr.	Weight addr.	Output addr.	Unused	
	`infsp`	0001	Input addr.	Weight addr.	Output addr.	First partial sum addr.	Unused
	`csps`	0010	Output neuron ID	Layer ID	First partial sum addr.	Unused	
Path Construction	`sort`	0011	Unsorted seq. start addr.	Seq. length	Sorted seq. start addr.	Unused	
	`acum`	0100	Input addr.	Output addr.	Cumulative threshold	Unused	
	`genmasks`	0101	Input addr.	Output addr.	Unused		
	`findneuron`	0110	Layer ID	Neuron position	Target neuron addr.	Unused	
	`findrf`	0111	Neuron addr.	Receptive field addr.	Unused		
Classification	`cls`	1000	Class path addr.	Activation path addr.	Result	Unused	
Others	Omitted for simplicity (`mov`, `dec`, `jne`, etc.)						

✿ Reducing Detection Cost: Selective Extraction

An orthogonal way to reduce the cost is to skip important neurons from certain layers altogether. In many networks, later layers have a more significant impact on the inference output than earlier layers (Raghu et al., 2017). Thus, one could extract important neurons from just the last a few layers to further reduce the cost (Chapter 3.7.6). When combined with forward extraction, this is equivalent to starting extraction later ("late-start"); when combined with backward extraction, this is equivalent to terminating extraction earlier ("early-termination"). This knob specifies the start/termination layer.

Summary The PTOLEMY framework provides three different knobs to explore the accuracy-efficiency trade-off. While the *extraction direction* applies to the entire network and hides the detection cost behind the inference cost, the *thresholding mechanism* and the *extracted layer* are specified at the layer level to reduce the detection cost.

```
1 def AdversaryDetection(model, input, θ, ϕ):
2   output = Inference(model, input)
3   N = model.num_layers
4   // Selective extraction only in the last three layers
5   for L in range(N-3, N):
6     if L != N-1:
7       // Forward extraction using absolute thresholds
8       ImptN[L] = ExtractImptNeurons(1, 1, ϕ, L)
9     else:
10      // Forward extraction using cumulative thresholds
11      ImptN[L] = ExtractImptNeurons(1, 0, θ, L)
12    dynPath.concat(GenMask(ImptN[L]))
13  classPath = LoadClassPath(argmax(output))
14  is_adversary = Classify(classPath, dynPath)
15  return is_adversary
```

Fig. 3.6: An adversarial detection algorithm expressed using the programming interface.

3.3.4 Programming Interface

PTOLEMY provides a (Python-based) programming interface that allows programmers to express a range of different algorithmic design knobs described above. Our programming interface is designed with two principles in mind, which we will explain using an actual detection algorithm expressed using the programming interface shown in Fig. 4.1.

Decoupled Inference/Detection The PTOLEMY programming interface decouples inference with detection, which allows programmers to focus on expressing the functionalities of the detection algorithm while leaving optimizations to the compiler and runtime. For instance, while the inference code (Line 2) and the path extraction code (Line 3–15) are expressed sequentially in the program, our compiler will understand that the program uses the forward extraction algorithm (Line 8 and 11), and thus will automatically pipeline inference with important neuron extraction across layers

(see Chapter 3.4.2).

Per-Layer Extraction Granularity Our programming interface provides the flexibility to specify the important neuron extraction method *for each layer* to leverage the three knobs described above to explore the efficiency-accuracy trade-off space. We will demonstrate its effectiveness in Chapter 3.7.6.

For instance in Fig. 4.1, the programmer selectively extracts important neurons only for the last three layers (Line 5). In addition, only the last layer uses the cumulative threshold to extract important neurons (Line 11), which is more accurate but requires more computations than using absolute thresholds, which is the method used by the other two layers (Line 8). Note that we do not allow backward extraction and forward extraction to be combined in one network to avoid discrepancies in the layer where they join.

3.4 ISA and Compiler Optimizations

This section describes how PTOLEMY efficiently maps detection algorithms expressed in the high-level programming interface to the hardware architecture. To that end, we first introduce the software-hardware interface, i.e., the Instruction Set Architecture (ISA) (Chapter 3.4.1), followed by the compiler optimizations (Chapter 3.4.2).

3.4.1 Instruction Set Architecture

PTOLEMY provides a custom CISC-like ISA to allow efficient mapping from high-level detection algorithms to the hardware architecture. The design principles of the ISA are two-fold. First, it abstracts away hardware implementation details; the semantics are closer to high-level DNN programmers, and the instructions will be decomposed by micro-instructions controlled by an FSM. Second, it exposes opportunities for compiler and hardware to exploit parallelisms.

The PTOLEMY ISA contains four types of instructions: *Inference*, *Path Construction*, *Classification*, and *Others*. They are high-level instructions in the CISC style that perform complex operations. We use a 24-bit fixed length encoding, and provide 16 general-purpose registers. Tbl. 3.1 summarizes the instructions. We highlight key design decisions.

- Inference These instructions dictate the inference process. In addition to support usual inferences (**`inf`**), PTOLEMY also provides an instruction that stores the partial sums to memory (**`infsp`**) during inference for backward extraction. Each inference instruction operates on one layer to match the per-layer extraction semantics in the high-level programming interface. Finally, the ISA also provides a special instruction that calculates and stores all the partial sums given an output feature map element (**`csps`**), which will be used by the compiler for memory optimizations.
- Path Construction This class of instructions is used to construct activation path dynamically at runtime for any given input. To construct path, the ISA provides instructions to identify important neurons (sorting **`sort`**, accumulate **`acum`**) and to generate the masks from the identified important neurons to form an activation path (**`genmasks`**). There are also instructions to calculate neuron addresses, which are convenient in finding the start address of a receptive field for a given neuron (**`findrf`**) and finding a given neuron given its position in the network (**`findneuron`**).
- Classification The classification instruction (**`cls`**) is used to classify an input as either adversarial or benign.
- Others The ISA provides a set of control-flow instructions (e.g., and **`jne`**), arithmetic instructions (e.g., **`dec`**), and scalar data movement instructions (e.g., **`mov`**).

Example Lst. 3.1 shows a sample code that uses cumulative thresholds to extract important neurons. Through a loop, it iteratively finds a receptive field (**`findrf`**), sorts

partial sums in the receptive field (**sort**), and uses the sorted partial sums to identify important neurons whose cumulative partial sums exceed the threshold (**acum**).

```
1  .set rfsize 0x200
2  .set thrd 0x08
3  mov r3, rfsize
4  mov r5, thrd
5  <start>
6  [update r7&r2 for next output neuron]
7  findneuron r2, r7, r4
8  mul r5, (r4)
9  findrf r4, r1
10 sort r1, r3, r6
11 acum r6, r1, r5
12 dec r11
13 jne <start>
```

Listing 3.1: Generating important neurons using a cumulative threshold. `.set` is a directive setting compiler-calculated constants. `[code]` indicates code omitted for simplicity.

It highlights an important design decision of the PTOLEMY ISA: all the detection related instructions use register operands. This design simplifies instruction encoding with little performance impact. For instance, the **findrf** instruction requires the receptive field size as an operand, which can be statically calculated by the compiler given the DNN model configurations. Since the receptive field size could be arbitrarily large and thus does not always fit in a reasonable, fixed-length encoding, a **mov** instruction is used to move the statically calculated immediate value to a register (`r3`), which is later used in the **sort** instruction. While a more complex instruction encoding that limits the range of immediate operands could eliminate this **mov** instruction, the performance overhead introduced by this **mov** instruction is negligible compared to the heavy-duty

sort and **acum** instructions.

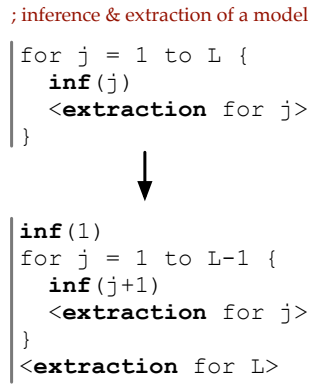
3.4.2 Code Generation and Optimization

The compiler maximizes performance by exploiting unique parallelisms and redundancies inherent to the detection algorithms. This is achieved through statically scheduling instructions, which minimizes runtime overhead and hardware complexity. Static scheduling is possible because the compute and memory access behaviors of both DNN inference and detection are known at the compile time.

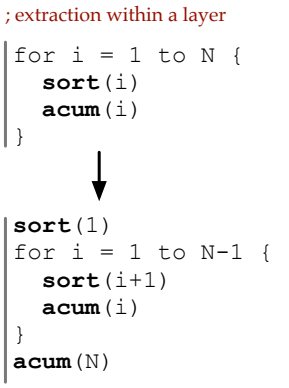
Layer-Level Pipelining A key characteristic of algorithms that use the forward extraction method is that inference and extraction of different layers can be overlapped. While the high-level programming interface decouples inference (INFERENCE) and extraction (EXTRACTIMPTNEURONS), and expresses them sequentially, our compiler will reorder instructions to enable automatic pipelining at runtime, in a way similar to classic software-pipelining technique (Allan et al., 1995).

Fig. 3.7a shows an example. We use pseudo-code to remove unnecessary details. <**extraction** `for j`> indicates the code block for extracting important neurons at layer `j`, and **inf** (j) indicates inference at layer `j`. By simply reordering instructions, inference of layer `j+1` and extraction of layer `j`, which are independent, could be pipelined. At the hardware level, once **inf** (j) is issued to execute on the DNN accelerator, <**extraction** `for j`> could be issued and executed immediately on our hardware extension (Chapter 3.5.2).

Note that our software pipelining technique does not fully hide the instruction latency to guarantee that a new instruction can be dispatched every cycle. Both inference and the extraction code block take tens of millions of cycles. Fully hiding latencies requires expensive optimizations in classic compiler literature (Triantafyllis et al., 2003; Hoste and Eeckhout, 2008). We find that our simple static instruction reordering is able to largely overlap inference with extraction, leading to very low performance overhead.



(a) Overlapping inference with extraction across layers in forward extraction. `L` is the total number of DNN layers.



(b) Neuron-level pipelining in important neuron extraction. `N` denotes the number of important neurons in the current layer's output.

Fig. 3.7: Pseudo-code of instruction scheduling examples. The code in (b) is the extraction block simplified in (a).

A side effect of not fully hiding the instruction latencies is that our hardware would still have the logic to check dependencies and stall the pipeline if necessary. But the hardware remains in-order without the expensive out-of-order instruction scheduling logic.

Neuron-Level Pipelining Similar to layer-level pipelining, our compiler will also automatically pipeline the extraction of different important neurons within a layer. Fig. 3.7b shows an example, where cumulative thresholds are used. The two steps needed to extract important neurons, sorting all the partial sums (**`sort`**) and accumulating the partial sums until the threshold is reached (**`acum`**), have data dependencies. The compiler overlaps the extraction across different important neurons (iterations), improving hardware utilization and performance.

Trading-off Compute for Memory Algorithms that use cumulative thresholds have high memory cost because all the partial sums must be stored to memory (Fig. 3.5).

However, if a receptive field does not correspond to an important neuron in the output feature map, its partial sums will not be used later. We observe that fewer than 5% of the partial sums stored are used later to extract important neurons.

We propose to use redundant computation to reduce memory overhead. Instead of storing all the partial sums during inference, we re-compute the partial sums during the extraction process only for the receptive fields that are known to correspond to important neurons in the output feature map. The compiler implements this by generating **`csps`** instructions to re-compute partial sums.

3.5 Architecture Support

This section introduces the PTOLEMY hardware architecture. Following an overview (Chapter 3.5.1), we describe the designs of major hardware components (Chapter 3.5.2 – Chapter 3.5.4).

3.5.1 Overview

Our architecture builds on top of a conventional DNN accelerator. Fig. 3.8 provides an overview of the PTOLEMY architecture, which consists of an augmented DNN accelerator, a Path Constructor that builds the activation path for an input, and a Controller that dispatches instructions, runs state machines that control the hardware blocks, and executes the final classifier. An off-chip memory stores all the data structures that are needed for inference and detection. Both the DNN accelerator and the Path Constructor use double-buffered on-chip SRAMs to capture data reuse and to overlap DMA transfer with computation. The controller's SRAM stores the compiled detection program and activation/class paths for classification.

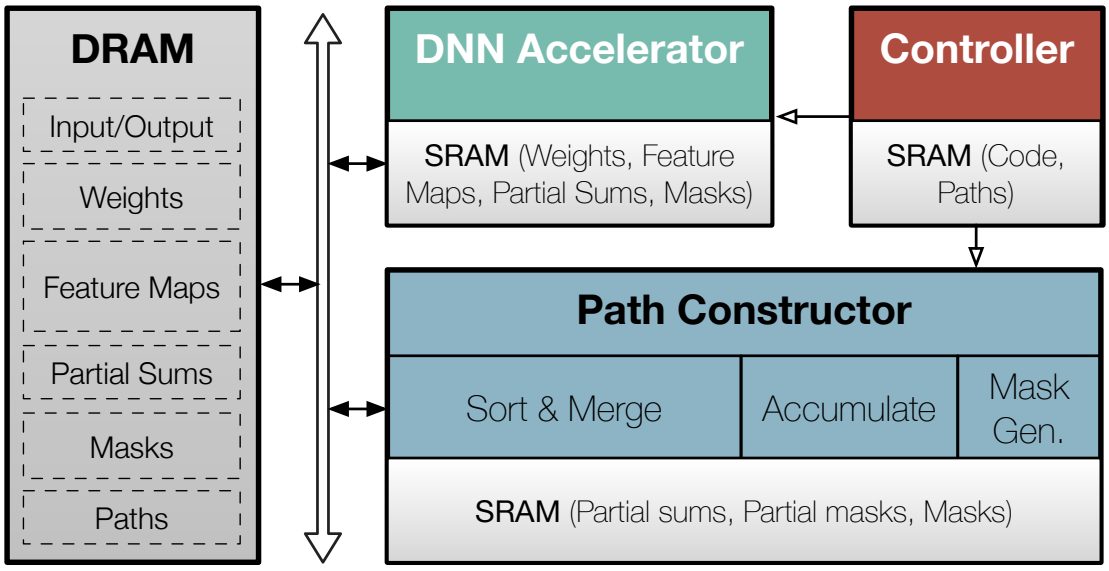


Fig. 3.8: PTOLEMY architecture overview.

3.5.2 Enhanced DNN Accelerator

PTOLEMY can be integrated into general DNN accelerator designs. Without losing generality we assume a TPU-like systolic array design (Jouppi et al., 2017). Each PE consists of two 16-bit input registers, a 16-bit fixed-point MAC unit with a 32-bit accumulator register, and simple trivial control logic.

PTOLEMY minimally extends each MAC unit. Fig. 3.9a shows the simple MAC unit augmentations (shaded). Specifically, algorithms that use absolute thresholds compare each partial sum with the threshold and store the single-bit mask to the SRAM; algorithms that use cumulative thresholds require each partial sum to be stored to the SRAM. Note that with the re-computation optimization, partial sums are recomputed at extraction time only for important neurons instead of being stored during inference.

To avoid the SRAM becoming a scalability bottleneck, the partial sums and the masks are double-buffered in the SRAM and doubled-buffered to the DRAM through a DMA. Later, the partial sums and/or masks are double-buffered back to the SRAM, similar to how feature maps and kernels are accessed. The extra DRAM space required

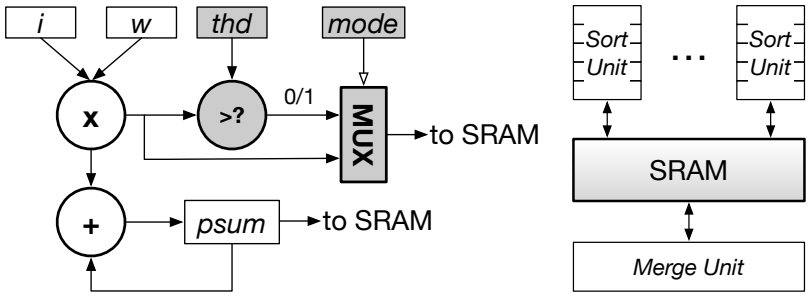
to store partial sums is small as we will show in Chapter 6.8.1. The additional DRAM traffic incurred by storing and reading partial sums is negligible ($<0.1\%$) compared to the original DRAM traffic since each partial sum is read and stored only once.

The PE array is used both for the usual inference and for re-computing partial sums as instructed by the **`clps`** instruction (Chapter 3.4.2). During re-computation, only the first row in the PE array is active because only a selected few elements in the output feature maps are to be re-computed.

3.5.3 Path Constructor

The goal of the path constructor is to extract important neurons and to construct activation paths. Algorithms that use cumulative thresholds requires sorting partial sums in receptive fields. Since receptive fields in modern DNNs are usually large (tens of thousands of elements), sorting all the elements on one piece of hardware could become a latency bottleneck as the sequence length increases. Our design splits a long sequence into multiple subsequences, which are sorted in parallel and merged together. Fig. 3.9b shows the sort unit organization. The sort unit uses the classic sorting network (Knuth, 2014), and the merge unit uses a standard merge tree, both have efficient hardware implementations (Mueller et al., 2012; Chen et al., 2015; Koch and Torresen, 2011).

The path constructor uses lightweight mask generation hardware, which generates the important neuron masks for each layer, from which the entire activation path (a bit vector) is constructed. The path constructor also integrates hardware that calculates similarities between an activation path and a canary class path, which is a highly bit-parallel operation. The SRAM in the path constructor is separate from the SRAM used by the DNN accelerator to avoid resource contention, and is also doubled-buffered.



(a) Enhanced MAC unit. (b) Sorting logic.

Fig. 3.9: Microarchitecture details. MAC and sorting constitutes 99.9% of the operations in our detection algorithm.

3.5.4 Controller

We assume a micro-controller unit (MCU) in the baseline hardware, as is common in today's DNN-based Systems-on-a-chip (SoCs) (xav). We piggyback two key tasks on the MCU: dispatching instructions and executing the final classifier to detect adversaries. Both are lightweight tasks that can be executed efficiently on an MCU without extra hardware.

Dispatching Instructions Thanks to the simple ISA encoding (Tbl. 3.1), the compiled programs can be interpreted on the MCU (i.e., software decoding) efficiently while avoiding extra hardware cost. The overhead of interpreting the code is negligible compared to the total execution time. The programs are very small in size. The largest one, which uses cumulative thresholds and backward extraction, is about 30 static instructions (below 100 bytes).

Classification The similarity between an activation path and the canary class path calculated from the path constructor is fed into a random forest (RF) for the final classification (Chapter 3.3.2). Our particular RF implementation uses 100 decision trees, each of which has an average depth of 12. In total, RF consumes about 2,000 operations on AlexNet (five orders of magnitude lower than inference), and could execute on an MCU in microseconds.

3.6 Evaluation Methodology

This section explains the basic hardware and software setup (Chapter 3.6.1) and the evaluation plan (Chapter 3.6.2).

3.6.1 Experimental Setup

Hardware Implementation We develop RTL implementation using Synposys synthesis and Cadence layout tools with Silvaco's Open-Cell 15nm technology (15n). The on-chip SRAM is generated using an ARM memory compiler and the off-chip DRAM is modeled after four Micron 16 Gb LPDDR3-1600 channels. We assume an ARM Cortex M4-like micro-controller (MCU) as the controller in the hardware (Chapter 3.5.4). The synthesis and memory estimation results are used to drive a cycle-level simulator for performance and energy analyses.

Networks and Datasets We evaluate PTOLEMY using two networks: 1) ResNet18 (Szegedy et al., 2017a) on the CIFAR-100 dataset (Krizhevsky et al., 2009) with 100 different classes and 50,000 training images, and 2) AlexNet (Krizhevsky et al., 2012) on the ImageNet dataset (Deng et al., 2009) with 1000 different classes and 1 million training images. The networks and datasets we evaluate are at the high end of the benchmark scale evaluated by today's countermeasure mechanisms (Carlini and Wagner, 2017a; He et al., 2017; Metzen et al., 2017), which mostly use much smaller datasets and networks (e.g., MNIST, CIFAR-10) (LeCun, 1998; Netzer et al., 2011) that are less effective in exercising the capability of our system. The test sets are evenly split between adversarial and benign inputs, following the common setup of adversarial attack research.

The clean AlexNet without attacks has an accuracy of 55.13% on ImageNet; ResNet18 has an accuracy of 94.49% and 75.87% on CIFAR-10 and CIFAR-100, respectively.

Attacks We evaluate PTOLEMY against a wide range of attacks. We first evaluate using five common non-adaptive attacks: BIM (Kurakin et al., 2016), CWL2 (Carlini and Wagner, 2017c), DeepFool (Moosavi-Dezfooli et al., 2016a), FGSM (Papernot et al., 2016b), and JSMA (Papernot et al., 2016b), which comprehensively cover all three types of input perturbation measures (L_0, L_2, and L_∞) (Akhtar and Mian, 2018).

We also specifically construct attacks that attempt to defeat our detection mechanism (a.k.a., *adaptive* attacks (Carlini and Wagner, 2017a)). In particular, we assume an adversary that has a complete knowledge of PTOLEMY's detection algorithms and the attacked model, and thereby generates adversarial samples by incorporating path similarities into the loss function.

Metrics We use the standard "area under curve" (AUC) accuracy metric (between 0 and 1) for adversarial detection (Huang and Ling, 2005), which captures the interaction between true positive rate and false positive rate. Unless otherwise noted, we report the average accuracy across all attacks. We confirm that the accuracy trend is similar across attacks.

3.6.2 Evaluation Plan

Our evaluation is designed to demonstrate that 1) PTOLEMY achieves similar or higher accuracy than today's detection mechanisms with a much lower performance penalty, and 2) the general framework allows for a large accuracy-efficiency trade-off. To that end, we develop and evaluate four algorithm variants using our programming model. All the compiler optimizations (Chapter 3.4.2) are enabled when applicable.

- BWCU: Backward extraction with cumulative thresholds.
- BWAB: Backward extraction with absolute thresholds.
- FWAB: Forward extraction with absolute thresholds.

- HYBRID: Hybrid algorithm where BWAB is used on the first half of a network and BWCU is used on the rest.

Baselines We compare against three state-of-the-art adversarial detection mechanisms: EP (Qiu et al., 2019), CDRP (Wang et al., 2018a), DeepFense (Rouhani et al., 2018). Both EP and CDRP leverage class-level sparsity. CDRP requires retraining and thus is not able to detect adversaries at inference-time. Note that we evaluate PTOLEMY using the exact same attacks used in the above papers.

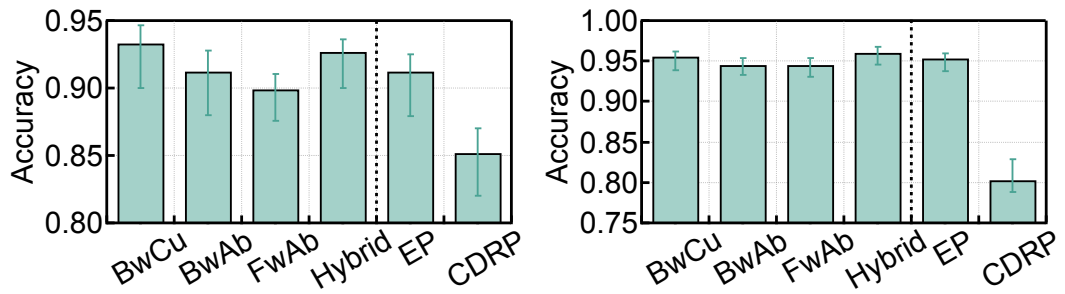
DeepFense represents a class of detection mechanisms that use modular redundancy. DeepFense employs multiple latent models as redundancies. We directly use the accuracy results reported in their papers. Note that DeepFense is evaluated using ResNet18 on CIFAR-10, on which we perform additional experiments for a fair comparison.

3.7 Evaluation

We first show the area and DRAM space overhead introduced by PTOLEMY's hardware extensions (Chapter 6.8.1) are small. We show that PTOLEMY provides more accurate detection (Chapter 3.7.2) with lower latency and energy overhead than prior work (Chapter 3.7.3 – Chapter 3.7.4). We show that PTOLEMY is robust against adaptive attacks that are specifically designed to defeat it (Chapter 3.7.5). PTOLEMY provides a large accuracy-efficiency trade-off space (Chapter 3.7.6). We further study the sensitivity and scalability of PTOLEMY (Chapter 3.7.7). Finally, we report additional results on several other models (Chapter 3.7.8).

3.7.1 Overhead Analysis

Area Overhead The baseline DNN accelerator incorporates a 20×20 MAC array operating at 250MHz. The accelerator has an SRAM size of 1.5 MB, which is banked at



(a) AlexNet on ImageNet.

(b) ResNet18 on CIFAR-100.

Fig. 3.10: Accuracy comparisons with EP and CDRP. Error bars indicate the max and min accuracies of all the attacks.

a 64 KB granularity. PTOLEMY augments the baseline hardware with a 32 KB SRAM banked at 2KB granularity for storing partial sums/masks, and a 64 KB SRAM used by the path constructor, which includes two 16-element sort units, one 16-way merge tree, and an accumulation unit. This accelerator is used in evaluating both PTOLEMY and all our baselines.

On top of the baseline DNN accelerator, PTOLEMY introduces a total area overhead of 5.2% (0.08 mm^2), of which 3.9% is contributed by the additional SRAM. The rest of the area overhead is attributed to the MAC unit augmentation (0.4%) and other logic (0.9%).

DRAM Space Under BWAB and FWAB, AlexNet and ResNet18 require 1.6 MB and 2.2 MB extra DRAM space. To show scalability, we also evaluated VGG19, which is 13× larger than ResNet18 and requires only 18.5 MB extra DRAM space. With the recompute optimization, AlexNet, ResNet18, and VGG19 require only an extra 12.8 MB, 17.6 MB, and 148.0 MB in DRAM, respectively under BWCU. The additional DRAM traffic is less than 0.1% (Chapter 3.5.2).

3.7.2 Accuracy

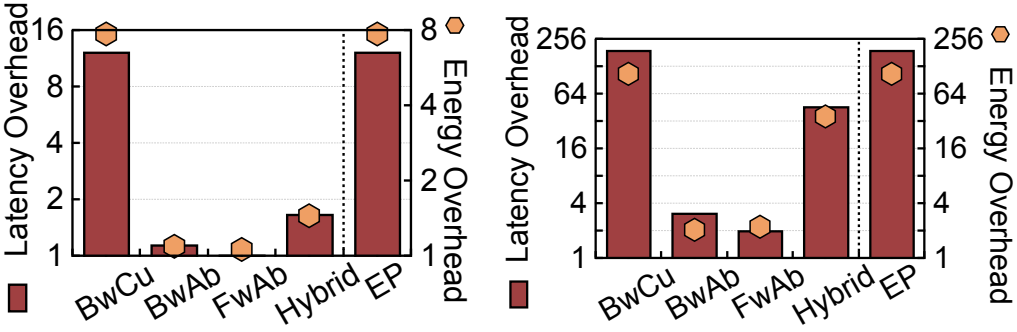
PTOLEMY's accuracy varies with the choice of θ and ϕ, which control the coverage of important neurons. Using BWCU as an example, Tbl. 3.2 shows how its accuracy changes as θ varies from 0.1 to 0.9. As θ initially increases from 0.1 to 0.5 the accuracy also increases, because a higher θ captures more important neurons. However, as θ increases to 0.9, the accuracy slightly drops. This is because a high θ value causes different class paths to overlap and become less distinguishable. Meanwhile, the latency and energy consumption increase almost proportionally as θ increases. We thus use $\theta = 0.5$ for the rest of our evaluation. The trend with respect to ϕ is similar, but is omitted due to limited space.

Table 3.2: Sensitivity of accuracy, latency, and energy of BWCU as θ varies. Latency and Energy are normalized to inference.

θ	Accuracy	Latency	Energy
0.1	0.86	4.7×	2.9×
0.5	0.94	12.3×	7.7×
0.9	0.91	25.7×	15.6×

PTOLEMY variants achieve similar or better accuracy than existing defense mechanisms. Fig. 3.10 shows the accuracy comparison. On AlexNet across all attacks (Fig. 3.10a), the three backward extraction-based variants (BWCU, BWAB, and HYBRID) outperform EP and CDRP by up to 0.02 and 0.1, respectively. FWAB uses forward extraction and has 0.03 lower accuracy than EP (0.06 higher than CDRP), indicating the accuracy benefits of backward extraction. On ResNet18 (Fig. 3.10b), PTOLEMY consistently achieves higher (0.14 – 0.16) accuracy than CDRP, and has similar or higher accuracy than EP (at most 0.01 accuracy loss).

Note that adversarial attacks generated by CWL2 have low confidence of the rank1 class, and the confidence of rank1 class is similar to that of the rank2 class. Thus, evaluating CWL2 let us understand PTOLEMY's robustness against adversarial attacks



(a) AlexNet on ImageNet.

(b) ResNet18 on CIFAR-100.

Fig. 3.11: Latency and energy comparisons with EP.

launched by "low-confidence" images. On Imagenet against CWL2, PTOLEMY's accuracy is 0.95, while the baselines are 0.94 (EP) and 0.85 (CDRP); on CIFAR10, PTOLEMY's accuracy is 0.96 while DeepFense is 0.93.

3.7.3 Latency and Energy

PTOLEMY could achieve low performance and energy overhead over usual DNN inference. Fig. 3.11a and Fig. 3.11b show the latency and energy consumption of the four PTOLEMY variants normalized to DNN inference, respectively. For comparison purposes, we also show the latency and energy of EP. We do not show the results of CDRP because CDRP requires retraining and is not suitable for online detection.

Although having the highest accuracy, BWCU also has the highest latency and energy overhead due to the expensive partial sum sorting and accumulation operations during extraction, which is serialized with inference. On AlexNet, BWCU introduces 12.3× latency overhead and increases the energy by 7.7×. The corresponding results on ResNet18 are 195.4× and 105.9×, respectively. The overhead on ResNet18 (18 layers) is higher than on AlexNet (8 layers), because as the network becomes deeper the amount of important neurons increases, which in turn increases the extraction time.

The overhead of BWCU is similar to EP, while BWAB, FWAB and HYBRID all

achieve much lower latency and energy overhead. BWAB uses absolute thresholds to avoid sorting and storing partials sums. BWAB reduces the latency and energy overhead on AlexNet to only 1.2× and 1.1×, respectively, and 3.2× and 2.0× on ResNet18, respectively.

FWAB further reduces the latency overhead to only 2.1% and 2.1× on the two networks, respectively, by using forward extraction to overlap extraction with inference. The latency overhead on ResNet18 is higher because ResNet18 is deeper with a higher important neuron density (explained above), leading to longer extraction latency that is harder to hide behind the inference latency. FWAB does not reduce energy overhead significantly comparing to BWAB, because it hides, rather than reducing, the amount of compute.

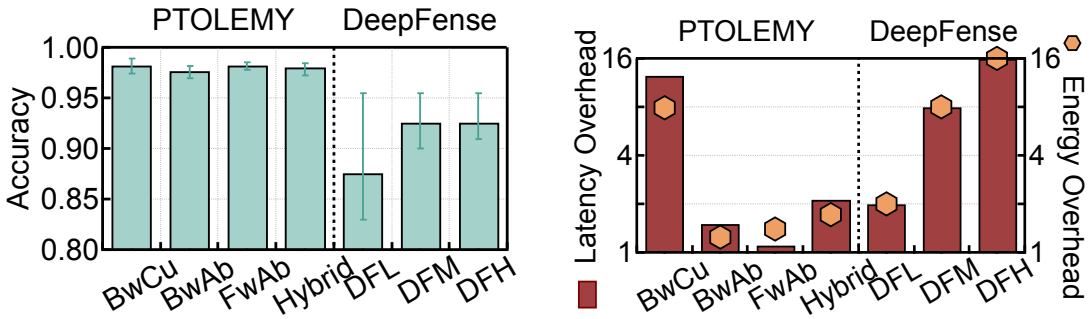
Finally, HYBRID provides a design point that balances efficiency with accuracy by combining cumulative thresholds and absolute thresholds. It leads to 1.7× latency overhead and 1.4× energy overhead on AlexNet, and the overheads are 47.3× and 36.1× on ResNet18, respectively.

3.7.4 DeepFense Comparison

We compare against the three default DeepFense variants, which differ in the number of redundant networks: 1 in *DFL*, 8 in *DFM*, and 16 in *DFH*. DeepFense is originally implemented on FPGA/GPUs; we perform a best-effort reimplementation on our hardware substrate for a fair comparison.

Fig. 3.12a shows the accuracy comparison between PTOLEMY and DeepFense using ResNet18 on CIFAR-10. All PTOLEMY variants achieve significantly higher detection accuracy over DeepFense. Specifically, FWAB, which has the lowest accuracy among all PTOLEMY variants, outperforms *DFH*, which is the most accurate setup of DeepFense, by 0.11 on average.

Fig. 3.12b shows the latency and energy of PTOLEMY and DeepFense variants nor-



(a) Accuracy.

(b) Latency and energy.

Fig. 3.12: DeepFense comparison.

malized to usual inference. With higher accuracy, BWAB and FWAB are also faster and consume less energy compared to all three DeepFense variants. For instance, FWAB reduces latency and energy overhead by 89.0% and 59.0%, respectively, compared with *DFL*, the most light version of DeepFense. The better efficiency of PTOLEMY over DeepFense indicates the effectiveness of exploiting the runtime behaviors of DNN inferences.

3.7.5 Defending Against Adaptive Attacks

Adaptive attacks refer to attacks that have complete knowledge of how a defense mechanism works and attempt to defeat that specific defense (Carlini et al., 2019; Tramer et al., 2020). We perform a best-effort construction of adaptive attacks against PTOLEMY, and show that PTOLEMY can effectively defend against adaptive attacks.

Constructing the Attacks To attempt to defeat PTOLEMY, we force an adversarial sample to have the same activation path as a benign input. However, since our path construction requires ranking/thresholding, which are non-differentiable, we opt for a differentiable approximation–a common practice in adversarial ML (Tramer et al., 2020). We experiment with several heuristics, and find that the most effective one is to force all the activations of an adversary to be the same as a benign input, i.e., a sufficient but

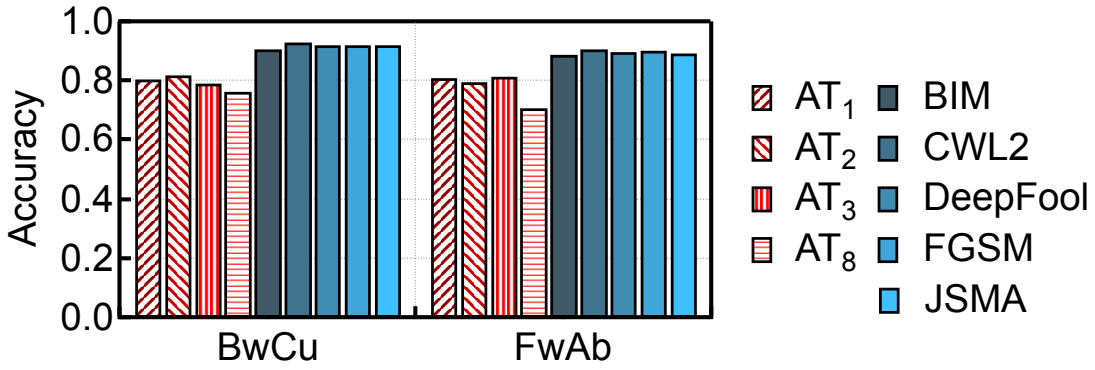


Fig. 3.13: Detection accuracy of PTOLEMY on various adaptive attacks (AT) compared to the five existing attacks.

not necessary condition.

Specifically, to generate an adversarial sample from an input x that has a true class c, we first randomly choose a benign input x_t of target class t from the training dataset, where $c \neq t$. We then add noise δx to x to generate x_a such that x_a's activations are as close to that of x_t as possible. This is achieved by minimizing the L2 loss $\sum_i \|z_i(x+\delta x) - z_i(x_t)\|_2^2$ as the objective function, where $z_i(\star)$ denotes the activations of $\star$ at layer i. To strengthen the attack, we choose five different x_t of different classes to generate five different x_a, and select the x_a with the smallest loss. We use ptolemyected gradient descent (PGD) (Madry et al., 2017) as the optimization method.

Results PTOLEMY detects these adaptive adversarial samples, even though they are generated specifically to "fool" PTOLEMY by having activation paths that are similar to their benign counterparts. Using AlexNet on ImageNet as an example, Fig. 3.13 shows the detection accuracy of BWCU and FWAB on the adaptive attacks (AT). AT_n denotes that activations of the last n layers are considered in the loss function when generating adversarial samples. Since AlexNet has 8 layers, AT_8 is the strongest adaptive attack. The detection accuracies on existing attacks are shown as for comparison.

Overall, the detection accuracy decreases as more layers are considered in generating the adaptive attacks, i.e., attacks become more effective. When only the first three

layers are considered by the adaptive attack, the adversaries are more easily detected by PTOLEMY than existing attacks. The detection accuracies on adaptive attacks are lower than those on non-adaptive attacks, confirming that adaptive attacks are more effective, matching the intuition (Carlini et al., 2019).

Validating and Analyzing the Attacks Our adaptive attack does not bound perturbation, i.e., is an unbounded attack. Following the guideline in Carlini et al. (Carlini et al., 2019) that "*The correct metric for evaluating unbounded attacks is the distortion required to generate an adversarial example, not the success rate (which should always be 100%)*", we verify the validity of our adaptive attack in two ways. First, we verify that the constructed attacks do reach 100% success rate; the average distortion, measured in Mean Square Error (MSE), is 0.007, and the maximum MSE 0.035.

Second, we show how the detection accuracy of PTOLEMY is impacted by the distortion rate introduced in the adaptive adversarial examples. The data is shown in Fig. 3.14, where every $< x, y >$ point denotes the average detection accuracy (y) for all the adaptive attacks whose distortions (MSE) is lower than or equal to a certain value (x). We find that overall the detection accuracy drops slightly as the distortion increases—an expected trend—although the trend is not strong, which is likely because the absolute distortion is too low (a desirable property) to demonstrate strong correlation with accuracy. We do verify that when the distortion is large enough to completely transform an image from one class to another, the detection accuracy would drop to 0, but at that point the input could not be considered an adversarial attack since the transformed image does not look like the original image.

We also investigate how the detection accuracy is impacted by the path similarities between the original class and the target class. We show the results in Fig. 3.15, where every $< x, y >$ point denotes the average detection accuracy (y) for all the adaptive adversarial inputs whose path similarity between the original class and the target class is lower than or equal to a certain value (x). While the path similarity between the

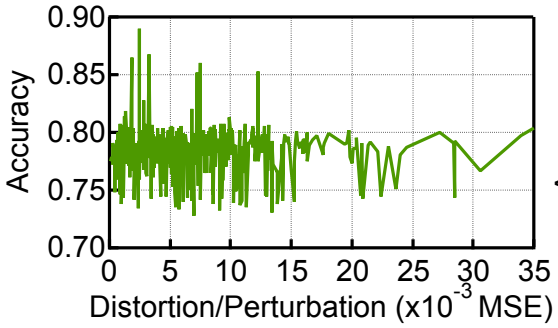


Fig. 3.14: Detection accuracy of adaptive adversarial inputs under different distortions.

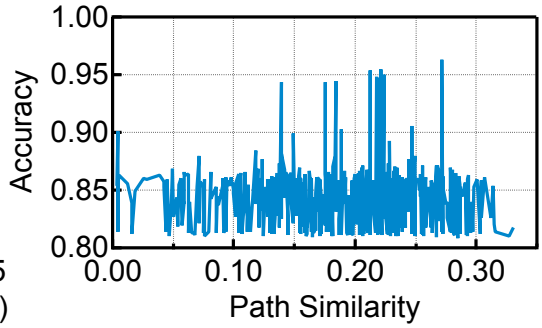


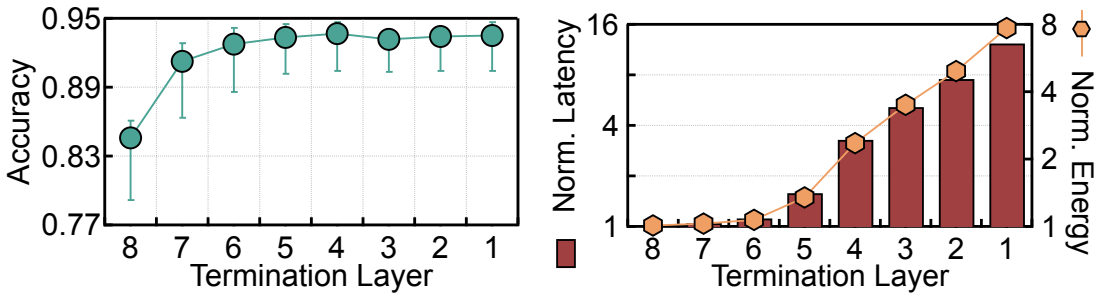
Fig. 3.15: Detection accuracy of adaptive attacks under different path similarities.

original class and the target class has a wide range (0.0 – 0.34), the detection accuracy does not correlate strongly with the path similarity. This is a desirable property, as it suggests that PTOLEMY is not more vulnerable when the attacker simply targets a similar class when generating the attacks.

Discussion The way we construct the adaptive attack is by approximating the hard path objective (i.e., forcing an adversarial sample to have the same activation path as a benign input) using a differentiable objective that constrains the individual activations. This relaxation let us formulate adversarial attack generation as an optimization problem that could be solved using effective optimization methods (e.g., PGD). If one were to force a hard constraint on the activation path, the objective function would not be differentiable.

In that case, a naive approach to generate adaptive attacks would be to exhaustively search all the possible perturbations. But without guidance such search would be prohibitively expensive (e.g., $(256^{3})^{40,000}$ for an 8-bit color depth, 200×200 resolution RGB image). We did try the exhaustive search method in a limited form, which generated results that add so much perturbation so that the resulted images do not look like the original images at all.

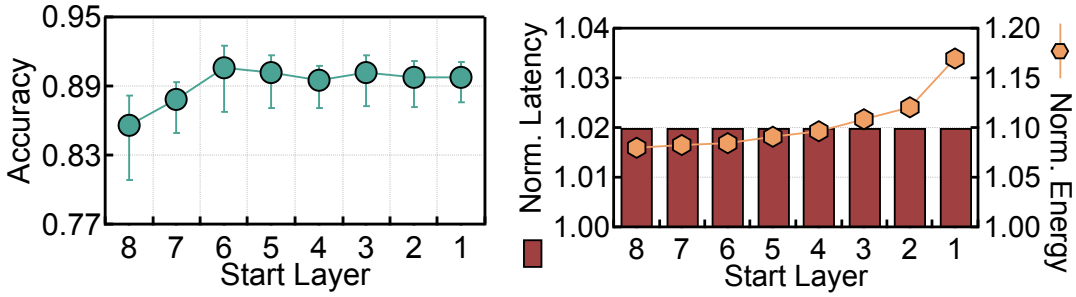
An interesting direction would be to investigate intelligent search heuristics (e.g.,



(a) Accuracy.

(b) Latency and energy.

Fig. 3.16: Accuracy, latency, and energy consumption under different termination layer in BwCu.



(a) Accuracy.

(b) Latency and energy.

Fig. 3.17: Accuracy, latency, and energy consumption under different start layer in FwAb.

simulated annealing) to find perturbations that meets the hard path constraint while fooling PTOLEMY. We leave this to future work.

3.7.6 Early-Termination and Late-Start

The PTOLEMY framework allows programmers to flexibly select which layers to extract important neurons from (Chapter 3.3.3). To trade accuracy for performance, programmers could start extracting important neurons later in forward extraction algorithms (as illustrated in Fig. 3.6), or terminate extraction earlier in backward extraction algorithms.

Early-Termination We use BwCu to showcase the trade-off that early-termination

in backward extraction offers. For simplicity, we show only the results on AlexNet; ResNet18 has similar trends. Fig. 3.16a shows how accuracy (y-axis) varies as the termination layer (x-axis) varies from 8 (the last layer) to 1 (the first layer). As AlexNet has 8 layers in total, terminating at layer 8 means extracting important neurons from only one layer. As extraction terminates later (further to the right on x-axis), more important neurons are captured and thus the accuracy increases. The accuracy increase eventually plateaus beyond layer 6, indicating marginal return of investment to extract more layers.

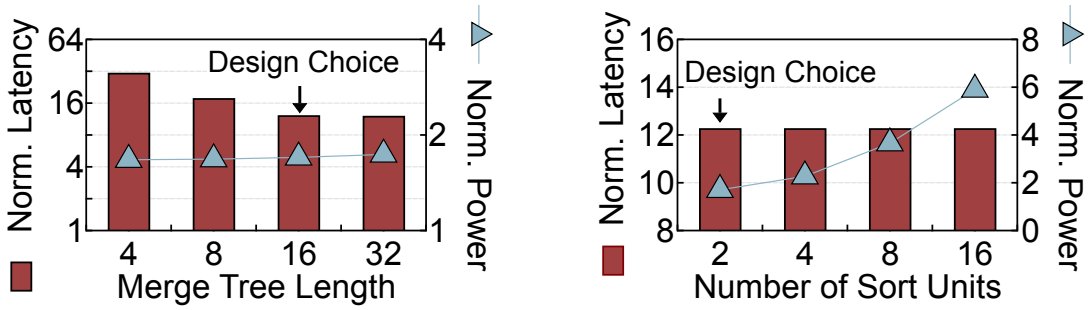
Fig. 3.16b shows how the latency and energy consumption varies with the termination layer. With virtually the same accuracy, extracting all the layers (i.e., terminating at layer 1) leads to 11.2× higher latency and 6.6× more energy compared to extracting only 3 layers (i.e., terminating after layer 6), which introduces only 1.1× and 1.1× latency and energy overhead over normal inference, respectively.

Late-Start We use FwAb as an example to demonstrate the trade-off that late-start provides to forward extraction-based methods. Fig. 3.17a and Fig. 3.17b show how the accuracy and latency/energy vary with the start layer, respectively.

Similar to early-termination, the accuracy increases as more layers are extracted, i.e., start earlier (further to the right). Interestingly, starting later does not help reduce the latency. This is because extraction latency is largely hidden behind the inference latency. However, starting later does reduce the energy consumption by 8.4% because less work is done.

3.7.7 Sensitivity and Scalability Studies

We show how PTOLEMY's performance varies with different hardware resource provisions in the path constructor. We report only the results of BwCu on AlexNet due to limited space. Fig. 3.18a shows how the latency and energy consumption (normalized to DNN inference) vary with the number of merge tree length (the number of partially



(a) Merge tree length.

(b) Number of sort unit.

Fig. 3.18: Performance vary with hardware resource.

sorted sequences that are merged simultaneously). As the merge tree length increases, the latency reduces (from 31.0× to 12.3×), but the power consumption stays virtually the same. This is because a 16-length merge tree contributes to only 2% of the total power.

Fig. 3.18b shows how the latency and power consumption vary with the number of sort units. We find out latency decreases only marginally with more sort units, because sorting is memory-bound and thus increasing computing units has a marginal impact. The power consumption, however, increases significantly, because the sort unit contributes significantly (33.4%) to the overall power in our design.

While our original DNN accelerator uses 16-bit precision, we also evaluate PTOLEMY under a 8-bit design. The area overhead increases from 5.2% to 5.5%. For AlexNet, the 8-bit design has 2.1% latency and 33.0% energy overhead using FwAb, comparable with 2.1% and 16.0% overhead of the original design. We also increase the MAC array size from 20×20 to 32×32. The area overhead increases from 5.2% to 6.4%. AlexNet has 4.4% latency and 16.4% energy overhead using FwAb, on par with the original 2.1% and 16.0% value.

3.7.8 Large Model Evaluation

On VGG16 (Simonyan and Zisserman, 2014) and Inception-V4 (Szegedy et al., 2017b), the average inter-class path similarity on ImageNet is only 41.5% and 28.8%, respectively, indicating that important neurons exist and class paths are unique in these models.

We also applied our detection scheme to DenseNet (Iandola et al., 2014), and achieved 100% detection accuracy with 0% false positive rate (FPR), higher than the previously best accuracy at 96% with 3.8% FPR (Ma and Liu, 2019). We use the detection accuracy and false positive rate instead of AUC in order to directly compare with the referenced method. We also evaluated ResNet50 on ImageNet using BwCu. The accuracy is 0.900, which is more accurate than EP (Qiu et al., 2019) (0.898).

3.8 Related Work and Discussion

Different mechanisms to counter adversarial attacks have been explored. One major class is to boost the DNN robustness at the training time through adversarial retraining (Bradshaw et al., 2017; Goodfellow et al., 2014a; Miyato et al., 2016; Gu and Rigazio, 2014), which incorporates adversarial samples into the training data. However, adversarial retraining does not have the detection capability at inference time. It also requires accesses to the retraining data, which PTOLEMY does not. PTOLEMY can also be integrated with adversarial retraining. Besides adversarial samples, we expect that PTOLEMY could also be used for detecting the execution errors of DNN accelerators caused by transient hardware errors (Leng et al., 2020, 2015; Papadimitriou et al., 2020).

Detection mechanisms have also been extensively explored, ranging from using modular redundancies (e.g., input transformation (Buckman et al., 2018; Guo et al., 2017; Thang and Matsui, 2019), multiple models (Rouhani et al., 2018), and weights

randomization (Dhillon et al., 2018; Xie et al., 2017)), to cascading a dedicated DNN to detect adversaries (Ma and Liu, 2019; Lu et al., 2017; Gong et al., 2017; Metzen et al., 2017). Wang et al. (Wang et al., 2020b) proposes to spatially share the DNN accelerator resources between the original network and the detection network. PTOLEMY differs from them in two ways. First, we show that using *path* as an explicit representation of the input, PTOLEMY can use a simple random forest classifier to detect adversarial inputs rather than complicated DNNs. Coupled with other performance optimizations, PTOLEMY provides very low (2%) overhead to enable detection at inference-time while others introduce several folds higher overhead. Second, PTOLEMY provides an algorithm design framework that allows programmers to make trade-offs between detection efficiency and accuracy.

Carlini et al. provides a checklist of best practices in evaluating defense mechanisms of adversarial attacks (Carlini et al., 2019). This paper exercises the following red teaming:

- Stated the threat model: attackers know everything (model, inputs, defense).
- Performed adaptive attacks (Chapter 3.7.5).
- Reported clean model accuracy (Chapter 3.6.1).
- Performed basic sanity checks (iterative attacks perform better than single-step attacks; increasing the perturbation budget strictly increases attack success rate; with "high" distortion, model accuracy reaches random guessing.).
- Analyzed success vs. distortion (perturbation) for our adaptive attack (Chapter 3.7.5).
- Showed that adaptive attacks are better (harder to be detected) than non-adaptive ones (Fig. 3.13).
- Showed attack hyper-parameters with the released code.

- Applied both non-adaptive attacks (covering all three types of input perturbation measures (L_0, L_2, and L_∞)) and adaptive attacks (Chapter 3.6.1).
- For non-differentiable components (in adaptive attacks), applied differentiable techniques (Chapter 3.7.5).
- Verified that the attacks have converged under the selected hyper-parameters.

4 A Reliable Perception System for Countering Adversarial Attacks

Deep neural networks (DNNs) are known to be vulnerable. Adversarial examples, which are carefully crafted examples, can easily lead deep neural networks to mispredict (Yuan et al., 2019; Madry et al., 2017; Moosavi-Dezfooli et al., 2016a; Li et al., 2019; Gao et al., 2018b). As deep learning has been widely used in mission-critical applications such as autonomous vehicle systems (Ramos et al., 2017; Rao and Frtunikj, 2018) and anti-fraud systems (Paula et al., 2016; Fang et al., 2021; Craja et al., 2020), the safety and robustness of deep learning systems countering adversarial examples have become more important than ever.

A primary countermeasure against adversarial attacks is adversarial example detection, as demonstrated in PTOLEMY. Leveraging the differences between the characteristics of standard and adversarial inputs, PTOLEMY effectively classifies adversarial examples from standard inputs. Nevertheless, a major limitation of adversarial example detection lies in its ability solely to detect but not process these adversarial examples.

The other main countermeasure of adversarial attacks is adversarial training (Papernot et al., 2016c; Miller et al., 2020; Tramèr et al., 2017b; Shafahi et al., 2019), which exposes the network architecture to adversarial samples at training time. Fundamentally, adversarial training trades off standard accuracy (SA) for robust accuracy (RA). The inherent SA-vs-RA trade-off has been observed, explored, and demonstrated in

Fig. 4.1: Standard model is vulnerable to adversarial samples; adversarially-trained model is more robust at a cost of standard accuracy; our method provides both high accuracy and robustness.

numerous studies (Zhang et al., 2019; Sun et al., 2019; Nakkiran, 2019; Raghunathan et al., 2019; Singla et al., 2021; Xu et al., 2021).

Fig. 4.1 illustrates the SA-vs-RA trade-off using an example where the traffic sign is adversarially-attacked (e.g., through a physical attack (Li et al., 2019; Sayles et al., 2021)). As a result, the standard model, while correctly classifies the car and the bike, mis-classifies the traffic sign into a traffic light. The adversarially trained model defends the adversarial attack on the traffic sign, but mis-predicts the bike, which is a standard input without contamination.

I argue that improving a network's robustness does not necessarily mean a significant drop in standard accuracy. I propose a generic framework that simultaneously improves the standard and robustness accuracy over prior adversarial learning schemes. Illustratively in Fig. 4.1, our framework would defend against the adversarial input (traffic sign) while also correctly predicting the standard inputs (the car and the bike). Fig. 4.2 shows a more comprehensive picture, where networks trained using the popular Projected Gradient Descent Adversarial Training (PGDAT) (Madry et al., 2017) under different configurations show a clear SA-vs-RA trade-off, while our frameworks

maintains both high SA and RA.

Our key insight is that a network's inference should automatically adapt to different inputs and to attacks of different strengths. Intuitively, a stronger attack requires a more robust model, and this selection must be done at inference time at a per-input basis. The key of the selection is an adversarial example detector like PTOLEMY. To that end, I propose a conditional adversarial learning framework, where the network architecture is parameterized with respect to a parameter λ that controls the standard-vs-robustness accuracy trade-off of the network. At inference time, λ is set for each input, allowing the network to adapt to the input dynamically.

Crucially, the condition parameter λ is learned from the input in a completely automated manner without any extra manual control from the users. This is accomplished by learning an input-specific λ such that λ represents whether an input is a standard input or an adversarial sample and, in the latter case, the perturbation level of the attack. I show that λ can be learned from the inherent inference behavior of an input. I then demonstrate an end-to-end trainable network that jointly learns λ and the subsequent inference conditioned on λ. I make three contributions in this work. First, I challenge the conventional wisdom that there necessarily is a trade-off between SA and RA. I introduce a conditional network architecture that adapts itself to the adversarial characteristics of an input and, thus, maintains both high SA and RA. Second, Unlike prior work on conditional adversarial learning (Wang et al., 2020a), I show how the condition parameter can be automatically inferred without user input. Prior work requires manual specification of the condition parameter, which is necessarily heuristic and, thus, lacks the ability to adapt to different inputs. Third, extensive experimental results demonstrate higher SA *and* RA compared to existing adversarial training schemes with just one model without retraining, ensemble, or extra user inputs.

4.1 Preliminaries and Motivation

4.1.1 Adversarial Attack and Defense

Deep learning models are vulnerable to adversarial attacks: small perturbations that fool the network (Szegedy et al., 2013). A range of different types of attacks exists depending on the attack model (Akhtar and Mian, 2018; Chakraborty et al., 2018). This paper targets the evasion attack, where the inference inputs could contain adversarial samples. Evasion attack is by far the most common attack form in practical settings (Hitaj et al., 2017; Madry et al., 2017; Moosavi-Dezfooli et al., 2016b; Ma et al., 2020; Li et al., 2020; Garcelon et al., 2020; Dolatabadi et al., 2020; Zhang et al., 2020b; Tramer et al., 2020)

The main countermeasure of adversarial attacks is adversarial training, where adversarial samples are exposed at training time (Goodfellow et al., 2014b; Lyu et al., 2015; Lin et al., 2020; Yang et al., 2020; Zhang et al., 2020a) Other methods such as gradient hiding (Papernot et al., 2017), feature enhancement (Xu et al., 2017; Levine and Feizi, 2020; Tian and Xu, 2021; Dusmanu et al., 2020) and adversarial example detection (Yin et al., 2021; Freitas et al., 2020) are also applied.

4.1.2 Adversarial Training

Adversarial training is one of the most studied countermeasures (Madry et al., 2017; Tramèr et al., 2017b). Most adversarial training methods increase the robustness of the network by introducing an extra robustness loss into the overall loss function:

$$\min_{\theta} \mathbb{E}_{(x,y)\sim\mathcal{D}}(1-\alpha) \times \mathcal{L}_c + \alpha \times \mathcal{L}_a, \tag{4.1}$$

where (x, y) is a pair of data and label drawn from distribution $\mathcal{D}$, f is the classifier parameterized by θ, $\mathcal{L}_c$ and $\mathcal{L}_a$ are loss functions over clean and adversarial samples

respectively, and α is a hyper-parameter controlling the model's ability to classify standard inputs vs. adversarial examples and it is usually fixed before training. The two loss terms are: $\mathcal{L}_c = \mathcal{L}(f(x;\theta), y)$, $\mathcal{L}_a = \underset{\delta \in \mathrm{B}(\epsilon)}{max}\, \mathcal{L}(f(x+\delta;\theta), y)$, where δ is the adversarial perturbation, and $\mathrm{B}(\epsilon) = \{\delta |\, ||\delta||_\infty \leq \epsilon\|\}$ is the allowed perturbation set with the pre-defined radius ϵ. Depending on the adversarial training strategy, the loss function $\mathcal{L}$ could be the cross-entropy loss (Cox, 1958), soft logits-pairing (Kannan et al., 2018), etc.

Standard-vs-Robust Accuracy Trade-off Many works (Raghunathan et al., 2019; Wang et al., 2020a; Yang et al., 2020) have shown that the inherent trade-off happens standard accuracy and robust accuracy exist, and the trade-off is dictated by α. In much of the prior work (Madry et al., 2017), α is statically set once in training. Using PGDAT (Madry et al., 2017) on CIFAR-10 as an example, Fig. 4.2 shows the trade-off between standard accuracy and robust accuracy. To increase the model's capability of countering adversarial examples, the accuracy on standard inputs drops as much as 9.5%.

Attack Strengths Another limitation of traditional adversarial training is the lack of ability to deal with adversarial samples with different attack strengths (perturbation levels). A model is typically trained with adversarial samples with only one attack strength and, thus, in theory, is able to defense attacks with that particular setting well. Simply training models by mixing different attack strengths leads to accuracy degradation.

We demonstrate the limitation of only using one model to defend attacks with different strengths in Fig. 4.3. "PGDAT" denotes a model trained using a typical adversarial training setting that mixes standard inputs and adversarial samples with a perturbation level $\epsilon = 6$. "PGDAT-Mixed" denotes a model trained by mixing standard inputs and adversarial samples with two different perturbation levels $\epsilon = 6$ and $\epsilon = 12$. "PGDAT-Dedicated" refers to dedicated models trained with the particular perturbation level same as the evaluation attack perturbation level. The y-axis shows the model accuracy

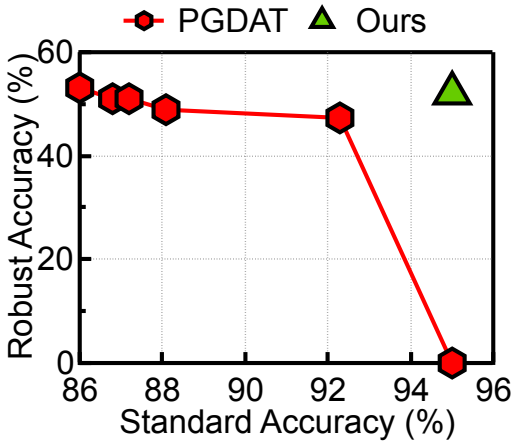


Fig. 4.2: Conventional adversarial training (PGDAT (Madry et al., 2017) on CIFAR-10 here) sacrifices standard accuracy for robust accuracy while our method achieves high robust accuracy without sacrificing standard accuracy.

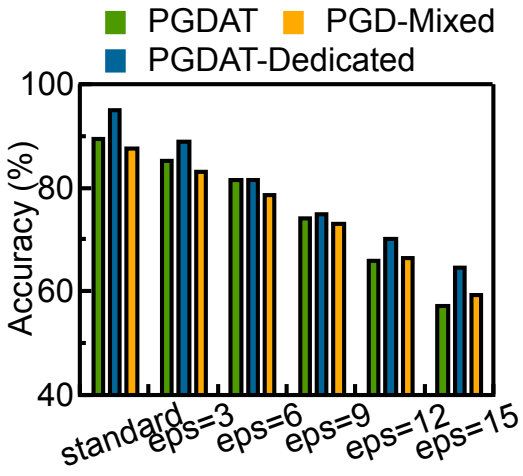


Fig. 4.3: Training one adversarial network to defend against different attack perturbations (eps) is less accurate than training a dedicated model for each attack perturbation level. Results are from PGDAT on CIFAR-10.

under standard inputs and adversarial samples of different perturbation levels.

PGDAT-Dedicated shows on average 3.7% accuracy improvement over PGDAT, which is in turn 1.8% more accurate than PGDAT-Mixed on standard inputs and, on average, 1.4% more accurate on adversarial samples of different perturbation levels. The results show that adversarial training by simply mixing different perturbation levels hurts accuracy. Training dedicated models, while generally more accurate, is undesirable as it increases the training effort and deployment complexity (Wang et al., 2020a).

4.1.3 Adversarial Sample Detection

Complementary to adversarial training which improves the model robustness at training time, one could also detect adversarial inputs when an input is available (Qiu et al., 2019; Metzen et al., 2017; Zhang and Zitnik, 2020). Traditionally, adversarial sample

detection techniques are generally used offline because they either are slow (Gan et al., 2020) or are fundamentally infeasible for inference-time detection, e.g., requiring accesses to training data (Wang et al., 2018b). Recent advancements have enabled fast and lean algorithms that can detect and reject adversarial samples at inference time (Aldahdooh et al., 2021) with high accuracy. For instance, Qiu et al. (Qiu et al., 2019) achieves a 96.0% Area Under Curve (AUC) accuracy on the CIFAR-10 dataset (Krizhevsky et al., 2009) and a 95.0% AUC on the CIFAR-100 dataset with negligible overhead.

A key contribution of our work is to integrate adversarial detection into the adversarial training process in such a way that the trained model is conditioned on the detection result. It addresses a crucial limitation of existing adversary detection techniques, where they do not have the ability to provide the correct output when an adversarial sample is detected, limiting their practical use.

4.2 Technical Approach

After providing an overview to our framework, we first focus on inference, describing how the condition is generated and how the condition is used to dynamically morph the network. We then discuss how a network is trained in our framework in the last subsection.

4.2.1 Main Idea Overview

Our main idea is to learn a DNN, whose inference is conditioned upon the adversarial characteristics of an input. In particular, we divide a N-layer network into two conceptual stages: a P-layer prefix stage used to generate the condition λ based on detecting the adversarial characteristics of an input, and an S-layer suffix stage, which contains different sub-networks, each of which is activated for a particular kind of input (e.g., standard vs. adversarial input) conditioned upon λ. Unlike prior conditional adversar-

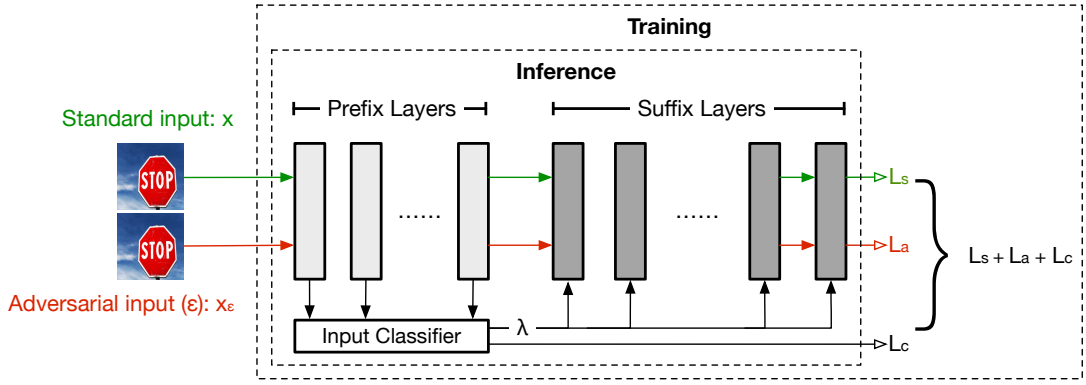


Fig. 4.4: MORPHADNET system overview. Prefix layers and input classifier is used to generate the condition λ. Suffix layers are dynamically conditioned upon λ. The input classifier in our implementation is a shallow 3-layer MLP.

ial learning (Wang et al., 2020a) that requires λ to be manually specified by users, λ in our framework is generated automatically without any user input. Fig. 4.4 provides an overview of our system at both the inference time and at the training time.

Inference The prefix layers serve two purposes. First, they are part of the inference network: the output of the prefix layer enters the suffix layer, just like how a normal inference model would work. Second, the prefix layers also extract adversarial characteristics of the input to generate the condition parameter λ, a scalar. In particular, λ takes 0 when the input is classified as a standard input and takes the value of the perturbation ϵ used to generate the adversarial sample if an input is classified as an adversarial attack. We will elaborate how the prefix layers predict λ in next subsection.

The suffix layers are dynamically adjusted given λ. Its design is inspired by multi-branch conditional learning work where the inference path changes based on the input (Bello et al., 2019). While a range of options are available, we use two particular kinds of modules to achieve adaptivity: dual batch normalization (Benz et al., 2021; Xie et al., 2020) and dynamic filtering (Jia et al., 2016).

Training The prefix and suffix stages are trained jointly end-to-end. The loss function is a weighted sum of three components: the loss on standard inputs, the loss

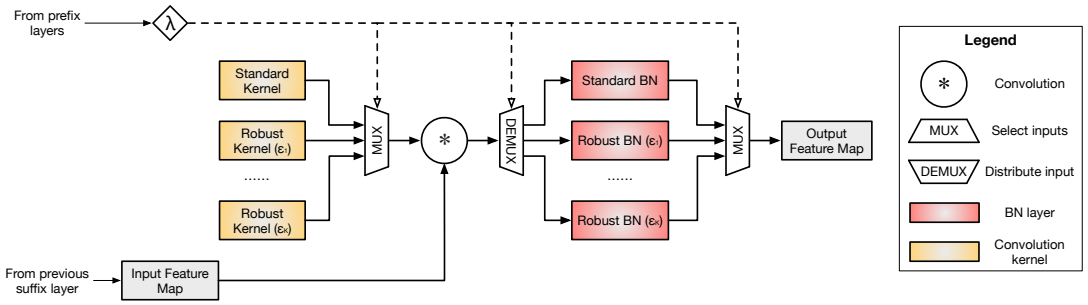


Fig. 4.5: The details of a dynamic suffix layer. We apply two techniques to achieve the configurability in dynamic layers which are dynamic kernels and K-BN layer.

on adversarial inputs, and the loss on the input classifier. Evidently, the first two losses depend on the third loss. We describe how this dependency is resolved and how training is practically implemented in training subsection.

4.2.2 Generating and Encoding the Condition

Encoding λ The goal here is to generate a λ that encodes the relevant adversarial information unique to a particular input. In particular, our current design ties λ to the perturbation level ϵ used to generate an adversarial attack. This is motivated by the observations that models specialized to specific perturbation levels are vastly different and significantly out-perform a generic network trained on different attacks (Fig. 4.3) (Tramer and Boneh, 2019). Thus, λ is encoded as a scalar value, which is 0 if the input is a standard sample and is equivalent to ϵ for adversarial inputs.

Learning λ Since λ is equivalent to the perturbation level ϵ, the goal is to predict ϵ of an input. In particular, we propose to predict ϵ by extracting and leveraging an input's inherent inference behavior in a network.

We find that a standard input of a class c tends to exercise a different "path" compared to adversarial inputs that are (mis-)predicted to have the same class c. A path $\mathcal{P}^{(x)}$ of an input x is defined as a collection of "important" activations and weights during the inference of x, where the importance is defined based on the activation magnitudes.

At training time, for each training sample we collect, layer by layer, all the activations (and their associated weights) that are above a pre-defined threshold θ. We then aggregate (union) the paths of all the traing samples of class c to create a class path $\mathcal{P}^c = \bigcup_{x \in \bar{x_c}} \mathcal{P}^{(x)}$, where $\bar{x_c}$ is the set of all the inputs of class c.

At the inference time, we construct a path $\mathcal{P}^{(x)}$ for each input x. We then generate a L-element vector $\mathcal{S}$ from $\mathcal{P}^{(x)}$ and $\mathcal{P}^c$, where L is the number of layers and c is the predicted class of x. Each element i in $\mathcal{S}$ characterizes the similarity between $\mathcal{P}^{(x)}$ and $\mathcal{P}^c$ at layer i. We use the Jaccard similarity coefficient encode the similarity:

$$\mathcal{S}_i = \frac{|\mathcal{P}_i^{(x)} \bigcup \mathcal{P}_i^c|}{|\mathcal{P}_i^{(x)} \bigcap \mathcal{P}_i^c|}, \tag{4.2}$$

$\mathcal{S}$ then enters a shallow, 3-layer MLP (the input classifier in Fig. 4.4) to predict ϵ. The intuition that $\mathcal{S}$ would be effective in predicting ϵ is that samples with a stronger perturbation tend to be more different from standard inputs than samples with smaller perturbations. $\mathcal{S}$ encodes the difference between an input's path and the corresponding class path. Thus, $\mathcal{S}$ is directly correlated with the the difference between an input and the standard inputs of the same class.

4.2.3 Dynamic Suffix Layers

The suffix layers are conditioned upon λ. Two extreme solutions exist. On one hand, one could use two completely different sub-networks that do not share weights; λ selects the sub-network. On the other hand, one could treat λ as an additional layer input and force all sub-networks to share (almost) all weights. The former is unscalable as it requires replicating the entire suffix layers multiple times, and the latter is inflexible, as the sub-network weights are dependent (e.g., up to an affine transformation if weights are conditioned on λ through an affine transformation (Perez et al., 2018)).

Our approach combines the best of both worlds selectively by instantiating different layer instances only in a small number of suffix layers. λ acts as a switch to choose be-

tween the layer instances. This architecture is illustrated in Fig. 4.5. Our design can be thought of as sharing the vast majority of weights across sub-networks but forcing the non-shared weights across sub-networks to be completely different. Clearly, how many layers are allowed to have the dynamic configurability dictates the trade-off between network overhead and accuracy, which we explore in evaluation.

Dynamic Knobs We resort to two common techniques to achieve the configurability in dynamic layers, shown in Fig. 4.5. First, we propose to use K mutually independent kernels in each dynamic layer to learn separate features from standard inputs and adversarial inputs with different perturbation levels, where K denotes the number of input classes. Second, we propose to leverage the phenomenon that statistics of standard inputs and adversarial inputs at batch normalization layers are distinctive (Xie et al., 2020; Wang et al., 2020a). We therefore replace BN layer in the network with a K-BN layer with K different BNs. At inference time, the network uses λ to select one of the K kernels and one of the K BNs, effectively conditioning itself on λ.

Scale-up the Network Traditional adversarial training methods are limited by the diversity of adversarial attacks they can deal with. This is because we usually have to train a dedicated network (with a unique training setting) for a given attack strength (ϵ). Mixing different attack strengths in one adversarial training setting leads to significant performance degradation (Fig. 4.3).

Our network provides two natural ways to scale to more attacks. First, if the input is an attack that has an ϵ that is beyond the K values seen at the training time, the prefix layers will predict a λ that is closest to the ϵ. Effectively, this allows us to adapt to unseen attacks by implicitly performing an interpolation on ϵ. Second, one could easily increase K to provide more coverage at the inference time. This is feasible as only a small portion (less than 1% of total parameters) of the network is selected to dynamic; thus, increasing K leads to little overhead.

Algorithm 1: Joint Training Process.

Input: Training set D; model f; maximal steps T; Adversarial example detector p; Weight on different loss $W_1, W_2, \ldots, W_K$

Result: Model parameter θ

for t = 1 *to* T **do**

Sample a batch of data (x, y) from D;

Create the corresponding adversarial samples: $x_{adv} = PGD(x)$;

Combine x and x_{adv} into a training batch x_r; predict the perturbation levels of inputs in prefix layers: $l_{adv} = p(x_r)$;

Calculate the adversarial sample classification loss: $Loss_{pre} = \mathcal{L}(l_{adv}, l_{gt})$;

According to the predictions, separate x_r into K batches, each with a predicted perturbation λ_i: x_{λ_i}, where $\lambda_i = 0$ denotes standard inputs;

Forward x_{λ_i} to the corresponding branch $P_{\lambda_i} = f(x_{\lambda_i}; \theta_{\lambda_i})$;

Calculate the loss on different branches $Loss_{\lambda_i} = \mathcal{L}(P_{\lambda_i}, y)$;

Calculate total Loss $Loss = W_0 \times Loss_{pre} + \sum_1^K W_i \times Loss_{\lambda_i}$;

Update model parameters $\theta_{\lambda_i} | i = 1, 2, \ldots, K$;

end

4.2.4 Training

We design an end-to-end training strategy to jointly train the prefix layers and the suffix layers. The training procedure is non-trivial because it must consider two unique aspects of our network. First, recall that the prefix layer serves two purposes: it is part of the overall inference network and also provides the path information to predict the λ of an input. As a result, the network loss must incorporate both inference loss and the λ prediction loss. Second, the exact suffix layers for a particular input depends on the prefix layers' prediction. We formulate the loss accordingly.

Algo. 1 shows the joint training process. For every batch of the data, we combine the standard images and corresponding adversarial images together and feed into the prefix

layers, which generate a λ for each input. Based on the λ value, we separate the training batch into K sub-batches, each with a unique λ value and enters the corresponding sub-network in the suffix layers.

The loss is combined from: $W_0 \times Loss_{pre} + \sum_1^K W_i \times Loss_{\lambda_i}$. $Loss_{pre}$ is the loss from the classifier predicting the perturbation levels and $Loss_{\lambda_i}$ is the inference loss of inputs that are predicted to have a perturbation λ_i (recall $\lambda_i = 0$ indicates an input is predicted as a standard input). We empirically set the weight of $Loss_{\lambda_i}$ to be ten times larger than that of $Loss_{pre}$. All losses mentioned in our application are cross-entropy loss.

4.3 Evaluation

We show the result of a basic two-branch setting on black-box attack including Transfer PGD attack and white-box attack including PGD attack and Auto attack. We compare with existing adversarial training mechanisms to improve SA-RA trade-off with manual or none test-time adaption. We also evaluate a three-branch setting. More detailed results about square attack and ablation study are presented in the supplementary material due to the space limitation.

4.3.1 Experimental Setup

Implementation The input classification in the prefix layers are implemented based on Ptolemy (Gan et al., 2020; Qiu et al., 2019), a state-of-the-art path-based adversarial sample detection algorithm. The original Ptolemy design supports only classifying an input as either standard or adversarial inputs. We augment its implementation to also predict the perturbation level ϵ. The original classifier in Ptolemy is a random forest, which we replace with a 3-layer MLP to be differentiable.

Table 4.1: Standard Accuracy (SA) and Robust Accuracy (RA) with a 2-branch MORPHADNET on Transfer PGD attacks. **PGDAT** is trained from with a mix of standard and adversarial samples whose perturbation (ϵ) is 8. **PGDAT-Dedicated** refers to training a dedicate network for a particular ϵ without standard samples. Ours-Oracle refers to an unrealistic version of MORPHADNET where the prefix layers have a 100% accuracy in predicting λ and, thus, can be seen as the upper-bound of our approach.

Dataset	Variants	SA	RA ($\epsilon = 2$)	RA ($\epsilon = 4$)	RA ($\epsilon = 6$)	RA ($\epsilon = 8$)
CIFAR-10	PGDAT	89.4%	85.3%	82.9%	81.9%	77.0%
	PGDAT-Dedicated	95.5%	91.9%	88.9%	82.0%	77.5%
	Ours-Strong	92.1%	89.1%	85.0%	82.0%	77.0%
	Ours-Weak	91.2%	88.9%	84.4%	82.8%	77.4%
	Ours-Oracle	95.0%	92.2%	90.1%	86.6%	77.2%
CIFAR-100	PGDAT	61.6%	57.8%	53.9%	49.9%	46.0%
	PGDAT-Dedicated	77.1%	67.8%	56.3%	52.1%	45.0%
	Ours-Strong	64.8%	63.7%	57.0%	53.1%	45.8%
	Ours-Weak	63.0%	59.3%	56.7%	52.1%	47.4%
	Ours-Oracle	72.1%	67.6%	63.6%	58.7%	45.7%

We evaluate two variants of our system. The stronger version Ours-Strong (Ours-S hereafter) uses the entire ResNet-18 as the prefix layers and ResNet-34 as the suffix layers. The weaker version Ours-Weak (Ours-W hereafter) uses the first 5 layers of ResNet-18 as the prefix layers. In both variants, only the first layer of *the entire network* is a dynamic layer (i.e., has two branches). This leads to a FLOPs increase of just 0.1%.

Attacks and Datasets We evaluate MORPHADNET against two attacks: the transfer PGD attack (Cheng et al., 2019) and the square attack (Andriushchenko et al., 2020). We generate transfer PGD attacks (Cheng et al., 2019) by attacking a ResNet-50 model. Both transfer PGD and square attacks (refer to supplementary material) in our evaluation are used as black-box attacks.

Developing white-box attacks for multi-branch, dynamic network is still an active area of research (Hu et al., 2020a): if an adversarial sample is sent to a different branch from what the sample was originally designed to attack (e.g., due to a mis-prediction

Table 4.2: SA and RA with a 2-branch setup on Square Attack.

Dataset	Variants	SA	RA ($\epsilon = 8$)
CIFAR-10	PGDAT	89.4%	80.3%
	Ours-Strong	92.1%	80.1%
	Ours-Oracle	95.0%	80.9%
CIFAR-100	PGDAT	61.6%	47.8%
	Ours-Strong	64.8%	46.2%
	Ours-Oracle	72.1%	46.7%

in the prefix layers), the attack effectiveness is significantly lower. We also try our best effort to create white-box attacks against MORPHADNET. For the suffix layers with more than one branch, we either select a branch to attack or attack a mix of them. We use PGD attack (Cheng et al., 2019) and AutoAttack (Croce and Hein, 2020) in the white-box attacks. We report both SA and RA in all cases.

Baselines We compared with three adversarial training mechanisms: the popular PGDAT (Madry et al., 2017), a recent adversarial training strategy Closer Look (Yang et al., 2020) which also aims to improve RA without sacrificing SA with dropouts in adversarial training, and OAT (Wang et al., 2020a), which provides a flexible SA-vs-RA trade-off by sampling different SA-vs-RA weights at training time and perform manual adaption at test time. For PGDAT, we also trained various PGDAT-Dedicated variants, each of which is trained using adversarial samples with only one particular perturbation level same to the evaluation attack perturbation level. The detailed configuration of each baseline will be described in the following sections.

4.3.2 Comparison with Adversarial Training on Black-box Attacks

The classifier we used to predict ϵ is able to correctly classify 75.4% of the standard samples and 93.5% of the adversarial samples on the largest ϵ level of 15. The detail discussion of the accuracy of the classifier can be found in the supplementary materials.

Tbl. 4.1 shows the result on transfer PGD attacks. On CIFAR-10, Ours-W outperforms the PGDAT baseline in both SA and RA. The baseline is trained using the standard adversarial training procedure by mixing standard inputs and adversarial inputs whose perturbation is 8. The SA improvement is 1.8%, and the RA improvements are between 0.4% and 3.6% for different perturbation levels. The same trend holds on CIFAR-100. Ours-S improves the SA by 3.2% compared to PGDAT, and the RA has an average improvement of 3.0%.

In general, Ours-S is more accurate than Ours-W, except when facing adversarial attacks generated using high perturbations (e.g., when $\epsilon = 8$), where the attack is so strong that the overall RA is generally low anyways. Ours-S introduces about 68.7% more FLOPs compared to Ours-W; they collectively present a speed-vs-accuracy trade-off to users.

To understand the upper-bound of our framework, we show the results of Ours-Oracle, which is a hypothetical variant of our framework, where the prefix layers are assumed to have an 100% accuracy in predicting the input ϵ. Ours-Oracle generally improves the SA and RA by 5.6% to 7.2% compared to the PGDAT baseline on CIFAR-10 dataset. The main reason that Ours-S and Ours-W fall behind Ours-Oracle is due to the mis-predictions in the prefix layers. We find that with a 0.8 AUC, the prefix layers in Ours-S mis-classified over 20% standard images into the adversarial category. This suggests that a better input predictor can readily improve the SA and RA with MORPHADNET.

Finally, we compare our framework to PGDAT-Dedicated. Not surprisingly, Ours-S and Ours-W have generally lower SA and RA than PGDAT-Dedicated. Ours-Oracle is

worse than PGDAT-Dedicated on SA, because when training Ours-Oracle the vast majority of the layers are forced to share weights to accommodate both adversarial samples and standard inputs. Ours-Oracle is competitive compared to PGDAT-Dedicated in RA settings, showing the ability to deal with different attack strengths.

Adaptive attack. To test the robustness of MORPHADNET under adaptive attack, we create the oracle adaptive attack against out input classifier. We assume that the adaptive attack can always break the input classifier and generate a wrong classification for the input, that is, to always use the opposite label the input classifier predict. For the attack with $\epsilon = 8$, we achieve a SA of 89.2% and a RA of 75.1%.

We test the model's ability on countering other black-box attacks to prove our method is not sacrificing robust accuracy for standard accuracy and we use square attack as an example. Square attack is a black-box attack which requires the model to be exposed to the attacker. As we described in Chapter 5.4.1, the multi-branch network architecture and runtime detector system design makes normal black-box attack improper for MORPHADNET. Hence, we incorporate PTOLEMY into the attack pipeline. For the oracle case of square attack, we apply attacks dedicated for the robust branch in our network. For the non-oracle case, we create an adaptive square attack targeting on breaking the defense of the entire MORPHADNET system. We process every image with PTOLEMY at every attack iteration to make sure the attack images will not be mis-classified into the standard branch. Same with baseline, both oracle and non-oracle attack is attacked with up to five hundred iterations and the attack perturbation amplitude is set to be eight.

Tbl. 4.2 shows the SA and RA of MORPHADNET on Square Attack. Ours-Strong achieves 2.7% and 3.2% SA improvements on CIFAR-10 and CIFAR-100, respectively. The RA is slightly lower in an acceptable range because the model is not specifically trained to defend square attacks.

Table 4.3: Standard Accuracy (SA) and Robust Accuracy (RA) on white-box PGD attacks and white-box AutoAttack (AA). **Attack on Robust Branch** refers to white-box attack against the robust branch of suffix layers. **Attack on Robust Branch** refers to white-box attack against the standard branch of suffix layers. **Ensemble Attack** refers to white-box attack against an ensemble of two.

Dataset	Variants	SA	RA: PGD	RA: AA
CIFAR-10	PGDAT (baseline)	89.4%	54.2%	42.9%
	Attack on Robust Branch (Ours)	92.1%	62.0%	44.8%
	Attack on Standard Branch (Ours)	92.1%	77.2%	77.5%
	Ensemble Attack (Ours)	92.1%	59.1%	78.8%

4.3.3 Comparison with Adversarial Training on White-box Attacks

We also evaluate our methods with white-box attack settings. We attack a selection of two branches (the robust one and the standard one) or a mix of both branches of the network and show the results in Tbl. 4.3. We perform both PGD attack and AutoAttack on suffix layers. Under different settings, MORPHADNET still achieves better RA than the baseline. Attacking either branch or an ensemble of them shows an average RA improvement of 11.9% on white-box PGD attack and 24.2% on white-box AutoAttack.

We further elaborate the unusual higher RA with the example of PGD attack on robust branch. Fig. 4.6 shows the comparison of SA (x-axis) and RA (y-axis). By changing the weight between standard loss and robust loss, PGDAT baseline shows the trade-off between SA and RA. Ours-Oracle process all the standard images with the standard branch and adversarial images with the robust branch, thus it pareto dominates the PGDAT baseline. When we use our classifier to replace the oracle selection and use blue marker to represent the accuracy. Ours-Strong shows less SA (2.4%) compared to Ours-Oracle as expected. Surprisingly, Ours-Strong also has a higher RA (4%).

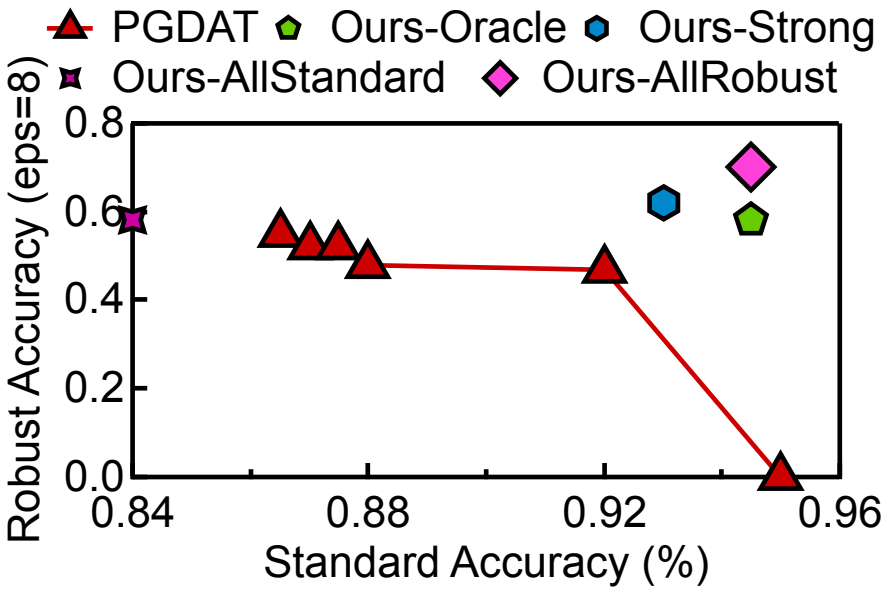
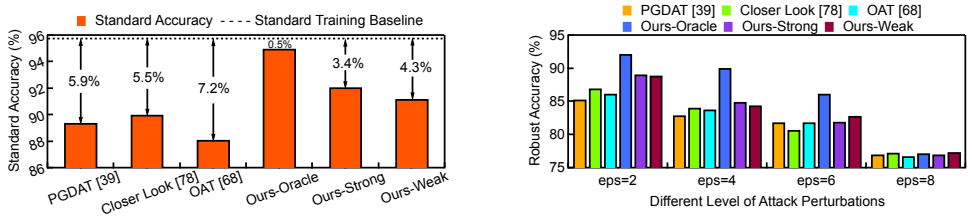


Fig. 4.6: Evaluation results on white-box PGD attack. *Ours-AllStandard* represents using only the standard branch for processing all images. *Ours-AllRobust* represents using only the robust branch.

The reason is that the white-box attacks are designed to defeat the robust branch, when the classifier mispredicts the images and uses the standard branch to process them, the attacks become a type of transfer (black-box) attack to the standard branch whose attack effectiveness gets degraded.

To further reason the behavior, we design another experiment to validate it. For the same set of evaluation images, we either use the standard branch only to process all the images (both standard ones and adversarial ones) or use the robust branch only. Theoretically, only using the standard branch should result in the same SA but higher robust accuracy because the attacks were designed to attack the robust branch, and using only the robust branch should result in the same robust accuracy but lower SA.

Fig. 4.6 use *Ours-AllStandard* and *Ours-AllRobust* to demonstrate the phenomenon. *Ours-AllStandard* has exact same SA compared to Ours-Oracle but with 12% higher robust accuracy. *Ours-Robust* has same robust accuracy compared to Ours-Oracle but with 11.5% less SA. The results again validates our conclusion drew from Tbl. 4.3.



(a) Standard accuracy drop compared to state-of-the-art methods.

(b) Robust Accuracy increase compared to state-of-the-art methods.

Fig. 4.7: Comparison with the state-of-the-art methods

4.3.4 Comparison with Existing Methods

We compare our work with two existing works, OAT (Wang et al., 2020a) and Closer Look (Yang et al., 2020), that target the trade-off between SA and RA by either manually doing selection at test time (OAT) or ignore test time selection (Closer Look). We re-implement Closer Look such that it can be evaluated on the same datasets and network as ours for a fair comparison. For OAT, we pick the λ to be 0.5 in the inference as it achieves both a high SA and a high RA.

Fig. 4.7a shows the SA under different schemes. As a reference, we show the accuracy under standard training (i.e., without adversarial samples). We see that Ours-S and Ours-W have significantly higher SA compared to the PGDAT, Closer Look, and OAT, which have a SA drop, from standard training, of 5.9%, 5.5%, and 7.2%, respectively.

Fig. 5.5b shows the comparison on the RA where the attack perturbation levels range from 2 to 8. Again, in virtually all cases, our algorithms out-perform the baselines. The average RA improvements of Ours-S over Closer Look and OAT are 0.9% and 0.9%, respectively. The RA improvement is more significant on small perturbations.

Table 4.4: SA and RA of a three-branch MORPHADNET on Transfer PGD attacks. **PGDAT** is adversarially trained from mixing standard inputs and adversarial samples with $\epsilon = 6$. **PGDAT-2eps** is a network trained from a mix of standard inputs, adversarial samples with $\epsilon = 6$, and adversarial samples with $\epsilon = 12$. **PGDAT-Dedicated** networks are trained with adversarial samples of a specific ϵ alone.

Dataset	Variants	SA	RA ($\epsilon = 3$)	RA ($\epsilon = 6$)	RA ($\epsilon = 9$)	RA ($\epsilon = 12$)	RA ($\epsilon = 15$)
CIFAR-10	PGDAT	89.9%	85.7%	81.9%	74.5%	66.4%	57.8%
	PGDAT-2eps	88.1%	83.5%	79.0%	73.6%	67.0%	59.7%
	PGDAT-Dedicated	95.5%	89.5%	82.0%	75.5%	70.6%	65.0%
	Ours-Strong	92.7%	88.3%	81.7%	76.6%	71.0%	65.0%
	Ours-Weak	90.4%	84.8%	80.0%	75.2%	69.1%	62.6%
	Ours-Oracle	94.2%	91.0%	81.5%	77.1%	71.1%	65.5%
CIFAR-100	PGDAT	62.3%	56.7%	48.2%	43.7%	37.9%	33.7%
	PGDAT-2eps	62.4%	50.6%	45.8%	41.9%	38.1%	34.6%
	PGDAT-Dedicated	77.1%	61.0%	52.1%	42.3%	37.9%	35.1%
	Ours-Strong	63.5%	58.7%	48.9%	43.2%	38.5%	34.8%
	Ours-Weak	62.8%	56.9%	48.7%	43.1%	38.4%	33.7%
	Ours-Oracle	67.9%	62.8%	48.3%	44.1%	38.9%	35.8%

4.3.5 Scalability

MORPHADNET can scale-up to integrate more dynamic branches as discussed in Chapter 4.2.3. We now evaluate a three-branch setting of MORPHADNET, where the first branch is trained to process standard inputs and the other two branches are trained to process adversarial inputs of perturbation levels $\epsilon = 6$ and $\epsilon = 12$, respectively, in our current implementation. For a fair comparison, we also augment conventional adversarial training such that it is exposed to both adversarial perturbation levels along with the standard inputs. We call this baseline PGDAT-2eps.

Tbl. 4.4 shows the results. Compared to PGDAT, PGDAT-2eps has sightly lower accuracy on adversarial examples with light but performs better on stronger attacks.

We evaluate five different perturbation levels (from 3 to 15) and show the results in Tbl. 4.4. Similar to the trend observed before, we out-perform PGDAT and PGDAT-2eps on both SA and RA. For instance, on CIFAR-10, the improvements is up to 7.5%

($\epsilon = 15$) on RA.

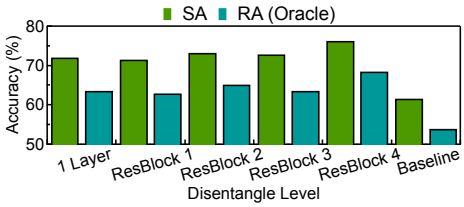
Ours-Oracle shows room for improvement. On CIFAR-10, the Ours-Oracle achieves 4.3% higher SA and on average 4.7% higher RA accuracy compared to PG-DAT. In addition, Ours-S is generally better than Ours-W, suggesting the speed-vs-accuracy trade-off.

4.3.6 Dual-BN Ablation and Sensitivity Results

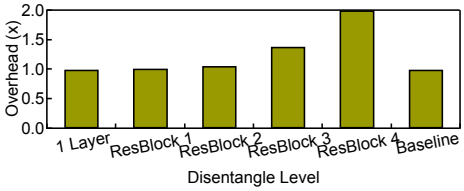
Dynamic batch normalization layers serve as a crucial role in the suffix layers. We perform an ablation study on experiments with and without dual-BN layers in the two branch suffix layers.

Sensitivity of Dynamic Layers . We perform more sensitivity study on the parameter and accuracy overhead of MORPHADNET. In the paper, we assume that only the first layer in the entire suffix network is a dynamic layer. This increases the total model parameters over the ResNet-34 baseline by less than 0.05% as shown by Fig. 4.8b. The network has a stronger capability to adapt to different inputs with more layers allowed to be dynamic as shown in Fig. 4.8a, the parameter overhead also increases. Fig. 4.8b shows this trade-off, where we further allow all layers in one or more residual blocks to be dynamic. When all four residual blocks are set to be dynamic, our overhead can reach to $2.0\times$ compared to the baseline.

Ablation study on dual-BN for CIFAR-10 dataset and CIFAR-100 dataset. We discuss the advantage of using multiple-BN layers. We provide the data on CIFAR-10 and CIFAR-100 dataset. Fig. 4.9a shows that using multiple-BN shows significant improvement on standard images and all attack perturbation levels. The SA is increased by 3.4% and the average RA is increased by 3.6% when dual BN is used. The results of CIFAR-100 dataset can be found in Fig. 4.9b. Results reveal that similar trends can be found in the CIFAR-100 dataset.

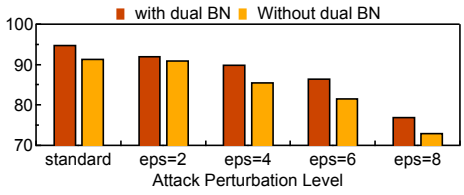


(a) Accuracy on enabling more layers to be dynamic.

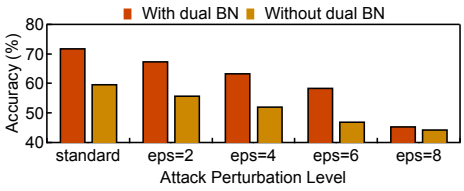


(b) Overhead on enabling more layers to be dynamic.

Fig. 4.8: SA and RA increase as more layers are allowed to be dynamic, but the number of model parameters also increases. The baseline here is a standard ResNet-34 model. "1 Layer" means only the first layer of the entire suffix network is dynamic. "ResBlock N" refers to variants where the first N residual blocks in the suffix network are dynamic.



(a) Ablation study on using dual-BN on CIFAR-10 dataset.



(b) Ablation study on using dual-BN on CIFAR-10 dataset.

Fig. 4.9: Comparison with manual test-time adaptation method.

4.3.7 Discussion on Existing Works

SA-RA trade-off problem is tricky. Most training methods will sacrifice standard accuracy when improving the robust accuracy. The reasonable way of improving the SA-RA trade-off is to increase the standard accuracy to the level of normally-trained network while still maintain the robust accuracy or improve it. MORPHADNET is the first work that conditions an adversarially-trained network based on the inference-time detection of input characteristics to solve the SA-RA trade-off, which is different from previous works. Related works either condition the network during training time or do not condition the network.

Condition the network during training time. One way to solve the SA-RA trade-

off during adversarial training is to enhance the loss function of adversarial training with weights (Madry et al., 2017). By tunning the weights between loss from standard examples (standard loss) and adversarial examples (adversarial loss), the model is able to trade standard accuracy for robust accuracy or vice versa. However, different weights on the loss term during training increase the number of models at inference time (each weighted loss during training corresponds to one individual model). OAT (Wang et al., 2020a) solves the multi-model problem by enhancing the process which not only tunning weights between standard loss and adversarial loss, but also condition the network based on the weighted loss during training. Although effective, OAT still has to manually pick the condition parameter during inference.

None conditional network. Another popular way of solving the SA-RA trade-off is to not putting any condition term during training and inference. Deep defense (Rouhani et al., 2018) integrates an adversarial perturbation-based regularizer into the training objective. Closer Look (Yang et al., 2020) introduces generalization methods such as drop out into different training methodology such as Locally Linear Regularization (LLR), Adversarial Training (AT), and Robust Self Training (RST) to improve both SA and RA.

5 Characterizing Inherent Fault-Tolerance Capabilities of Autonomous Machine Software

Autonomous machines are not reliable. With the rapid growth of the autonomy inside drones, robotics and vehicles (Kalra and Paddock, 2016; Koopman and Wagner, 2017; Yu et al., 2020), the unreliability of hardware and software in autonomous machines have also resulted in many crashes of the systems. Among them, the reliability of autonomous vehicles, which are vehicles executing driving tasks autonomously, is crucial, as faults that happen in autonomous vehicle systems could lead to severe consequences (Boudette, 2021a,b).

Despite numerous efforts in improving the safety of AV products (Moody et al., 2020; Reschka, 2016; Gan et al., 2020), a myriad of sources threatening AV safety still exist. A single bit-flip of the transistors inside the hardware caused by thermal irregularities or cosmic rays (Mukherjee et al., 2005; Dixit and Wood, 2011; Zhang and Shanbhag, 2006) can result in a silent data corruption (SDC) of an arbitrary algorithm in the AV software stack. The SDC can propagate to the following software and influence the behavior of the vehicle. Carefully designed adversarial attacks on traffic signs and traffic lights are proved to be able to mislead the perception module of the AV (Komkov and Petiushko, 2021; Guo et al., 2021; Lengyel et al., 2021) easily. Soft-

ware bugs (Garcia et al., 2020; Koopman and Wagner, 2016) in the AV software stack can also be the source of unreliability.

Existing fault-tolerance techniques are expensive. Traditional protection mechanism such as modular redundancy (Abraham and Siewiorek, 1974; Engelmann et al., 2009; Kim and Shanbhag, 2010), anomaly detection and recovery (Ahmed et al., 2016; Chandola et al., 2009; Patcha and Park, 2007), and re-execution (Kim et al., 2010; Roth, 2005) introduce spatial and/or temporal overhead, challenging the real-time nature of AV.

In the preceding two chapters, I introduced PTOLEMY, an efficient method for adversarial example detection, and MORPHADNET, a dynamic framework for adversarial training that simultaneously enhances standard and robust accuracy. Both approaches aim to develop a reliable perception module for the computing systems used in autonomous machines. Nevertheless, the software stack of autonomous machines comprises more than just a perception module. Additional critical components include localization, planning, and control, all of which contribute to the overall performance of such systems.

In this chapter, I propose BRAUM, a dynamic protection system rooted in the understanding of the inherent fault tolerance mechanisms of autonomous software. Using the popular Autoware AV software as a case-study (Kato et al., 2018), I observe that many algorithms in an AV software have inherently error-masking and/or error-attenuation capabilities through operations such as operator union and low-pass filtering. BRAUM systematically identifies these error-masking mechanisms and leverages these mechanisms to selectively provide error protection to AV software with minimal overhead.

To identify the error-masking capability of AV algorithms, I conduct a large-scale fault injection into the AV software stack and use program analysis techniques to trace how faults propagate and are masked by each algorithm in the AV software. Our fault injection framework synthesizes and injects faults that mimic different forms of potential faults an AV software could encounter, including transient errors, adversarial

attacks, and software bugs. The fault injection framework is lightweight so as to minimize the impact on the behavior of the original software stack.

Our fault injection and analysis reveal many mechanisms for error masking and/or error attenuation possessed by different algorithms in the AV software stack. Building on top of these inherent mechanisms, I propose the notion of Fault Tolerance Level (FTL), which describes whether and how the output error of an individual algorithm will be masked before reaching the actuator. I show an iterative algorithm that calculates the FTL of an algorithm in an AV software stack.

Leveraging the FTL calculation, I propose a dynamic protection system that selectively elides and/or relaxes protection of certain algorithms in the software to demonstrate BRAUM is able to help reduce protection overhead. Intuitively, if an algorithm has a "high FTL", e.g., its output errors can always be masked before reaching the actuator, there is no need to, for instance, re-execute that node to avoid soft errors.

In summary, I make the following contributions in this chapter. First, I propose a lightweight fault injection framework to synthesize and inject representative faults that an AV could encounter and analyze the fault injection results to identify inherent error masking/attenuation mechanisms in AV software. Second, I propose the notion of FTL to intuitively describe how *invulnerable* an algorithm is, and provide an iterative algorithm to calculate the FTL of each algorithm in the software stack. Finally, I design a simple dynamic protection system that selectively elides/relaxes protection for high-FTL algorithms to illustrate the idea of BRAUM. I implement our protection system on Autoware, a widely-used AV software. Experimental results show that we reduce error propagation rates by 90.1% compared with a baseline without any protection and reduces the execution overhead by 47.2% compared to a baseline that provides always-on protection.

5.1 Background

We briefly introduce the design of AV software in Chapter 5.1.1, the source of threats to the reliability of AV software in Chapter 5.1.2 and existing protection mechanism in Chapter 5.1.3. We discuss common protection methods in Chapter 5.1.3.

5.1.1 Autonomous Vehicle Software Stack

Fig. 5.1 shows a high-level overview of an AV system which consists of a computing system and a mechanical system. At each time frame, the computing system takes input from the sensors (e.g., cameras, LiDAR, GPS) to infer an actuation (e.g., throttle, brake, steering wheel angle) to the actuators and further control the mechanical components of the AV.

The computing system runs a complicated software system on the hardware. Typical AV software stacks such as Autoware (Kato et al., 2018) and Baidu Applo (Apo) perform three major tasks that enable the autonomous driving capability: perception, localization and control. The perception module detects and interprets the environments with the information provided by the cameras and LiDARs. The localization module locates the current position of the AV on the map with the input of GPS and IMUs. Planning digest both results from perception and localization module to plan for the trajectory and control the actuators. Usually each module contains multiple algorithms. Different algorithms are connected in a consumer-producer fashion, formulating a complicated computational graph.

5.1.2 Source of Threats to AV Software

AV software is not running in a safe and reliable environment. In reality, when the vehicle is driving autonomously on the road, different sources of threats exist. We briefly describe the three main error sources that this paper focuses on.

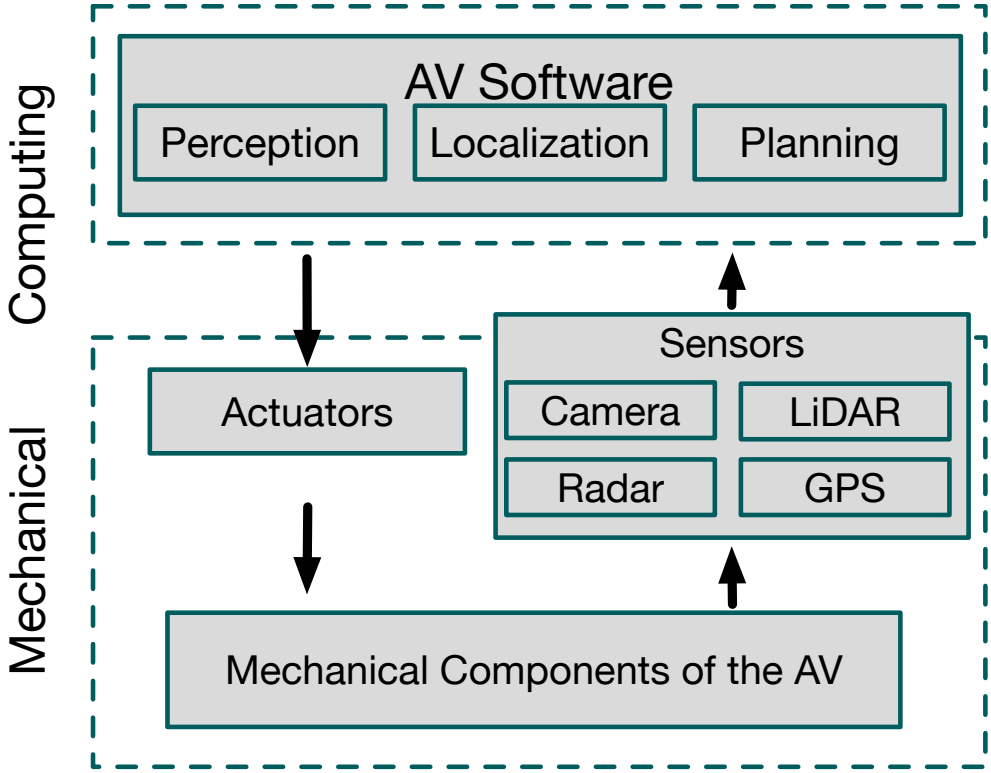


Fig. 5.1: An overview of AV system.

Soft Errors. Soft error is a type of error that changes the states of a logic device (e.g., a SRAM cell). A soft error can be caused by cosmic rays or thermal impact. Multiple works (Iturbe et al., 2016a; Blome et al., 2005; Mukherjee et al., 2003) have shown that a bit-flip caused by the soft error can easily result in incorrect outputs of a program, i.e., Silent Data Corruption (SDC). SDC can take place in stage/algorithm in an AV software stack. An SDC occurring at one stage could propagate and eventually corrupt the output of the entire software stack, crashing the vehicle (Bandeira et al., 2019).

Adversarial Attacks. Unlike soft error and software bugs that take place unintentionally, adversarial attacks are carefully crafted manipulations of the input of certain algorithms that cause an algorithm to misbehave. The most well-studied example is

the adversarial attack on the input image to the deep learning-based perception module. Negligible perturbations to an input image can lead a perception DNN to mispredict, e.g., recognizing a stop sign as a green light (Yuan et al., 2019; Madry et al., 2017; Li et al., 2019). Adversarial attacks have also been extended to other sensor input such as point cloud data (Liu et al., 2019a; Zhang et al., 2021).

Software Bugs. Software can be buggy. Even with experienced programmers and extensive testing techniques, AV software is vulnerable to different kinds of bugs. Previous work (Garcia et al., 2020) found 499 bugs in the two widely-used AV Software Autoware (Kato et al., 2018) and Baidu Applo (Apo). These bugs exist in the perception, localization, planning and actuation of the AV software. 10.6% of the bugs will lead to a crash of the AV system in the end (Garcia et al., 2020), showing that software bugs can be a serious threat to the safety of AVs.

5.1.3 Common Protection Mechanisms

Redundancy. Redundancy, both temporally and spatially, serves as an effective way of countering the threats to the AV software. Temporal redundancy refers to executing a part of the code more than once (Kim, 1999; Rivers, 1998; Oplinger and Lam, 2002). Redundant executions can help alleviate the threat of SDC caused by soft errors as they are transient. Temporal redundancy introduces significant performance overhead. Executing each software module twice effectively halves the performance.

Spatial redundancy refers to executing the same algorithm using different physical hardware instances (Lyons and Vanderkulk, 1962; Kastensmidt et al., 2005). For instance, Tesla's Full Self-Driving (FSD) chip makes two copies of the entire processing logic, effectively introducing a dual modular redundancy (Pete Bannon, 2019). Modular redundancy has been shown to be effective against software errors (Yim et al., 2012) and adversarial attacks (Rouhani et al., 2018). Modern processors usually provide hardware support to minimize the performance overhead of executing on identical hardware

copies (de Oliveira et al., 2017, 2018). As a result, the main overhead of special redundancy comes from the added silicon area and the associated non-recurring engineering costs, which are expected to increase as AV platforms are increasingly integrating specialized accelerators (Gan et al., 2021; Liu et al., 2021).

Anomaly Detection. As an alternative to redundancy, anomaly detection shields the errors at the sources. Unlike much traditional software, Av software process temporal inputs, i.e., sequences of sensors inputs, which exhibit strong temporal consistency. For example, when a car is driving in a straight lane, it is unlikely that the path planning module will issue a sudden acceleration to the actuator. Therefore, errors in AV software sometimes are manifested as outliers that break the temporal consistency. Different techniques on anomaly detection (Sample and Schaffer, 2013; He et al., 2016) are proposed to detect outliers in AV software. Anomaly detection, however, introduces overhead due to the execution of the detection algorithm.

5.2 Understanding Inherent Fault Tolerance in Autonomous Machine Software Stack

We first describe our error injection methods (Chapter 5.2.1). We then analyze how the fault propagates (Chapter 5.2.2) to identify different masking mechanisms (Chapter 5.2.3). From individual nodes' masking mechanisms, we describe how the fault-tolerance level of each node is derived (Chapter 5.2.4).

5.2.1 Error Definitions and Injection

Autoware is a widely used AV software stack. We apply Autoware into the simulation platform of CARLA (Dosovitskiy et al., 2017) to form realistic AV driving scenarios. Autoware is built with the support of Robotic Operating System (ROS), where different algorithms in Autoware is represented by a separate process or ROS node. Different

ROS nodes communicate through ROS messages which is the output of each algorithms. All the ROS nodes and ROS messages formulate a large directed graph which we will refer to ROS graph in the following context. The ROS graph is a strict equivalence of Autoware. The error injection happens on our server with 8 Intel(R) Xeon(R) W-2123 CPUs and a Nvidia Quadro RTX 4000 GPU and is tested on the Ubuntu 18.04.5 system.

The goal of BRAUM error injection framework is to mimic different kinds of errors AV software may encounter while being lightweight enough to not influence the regular execution of AV software. We achieve this by injecting three types of errors in Autoware.

Soft errors. To mimic soft error-induced SDC, we leverage architectural-level register error injection. During BRAUM error injection, the selected victim ROS node will send out its process ID (PID) to the error injection process so that the injection process can attach to the victim ROS node via *ptrace* system call. The `ptrace` system call allows us to manipulate the register files of the selected victim ROS node. First, we randomly pick a general-purpose or floating point register and randomly pick a bit to flip. Second, we use `ptrace` and the PID of the running process to pause the execution, obtain the register value, inject the fault, and resume the execution. This architecture-level register error injection has little overhead, as shown in previous error injection tools (Porpodas, 2019; Hsiao et al., 2021).

Adversarial attacks. To mimic potential adversarial attacks, our strategy is carefully corrupt the *output* of relevant ROS nodes. For the perception module, which mainly performs object detection and tracking, we emulate two common types of adversarial attacks (Madry et al., 2017; Yuan et al., 2019), non-targeted attack and targeted attack. For non-targeted attacks, we randomly change the detected object class to another class that exists in the dataset. For targeted attacks, the detected object class is randomly changed to another class commonly seen in AV (e.g., person, stop signs). To emulate corruptions in bounding boxes, we either create a very large bounding box

(i.e., 240×240 pixels) when the ground truth is an empty box or remove the bounding box altogether if otherwise.

Adversarial attacks on the localization and planning modules are much less common in literature. We create our best-effort localization and planning attacks by assuming that an attack on localization moves the vehicle position away from the ground truth anywhere between 5m and 300m, similar to prior work (Patil et al., 2021; Wang et al., 2022). Attacks on planning are similarly emulated except we move the predicted future, rather than current, position of the vehicle.

Software bugs. To emulate software bugs, a randomly generated error is applied (added) to a node's output signal. Both adversarial attacks and software bugs are implemented by using the ROS topic publish mechanism with little overhead.

Summary. In total we cover 23 ROS nodes (and 26 ROS topics) as shown in Tbl. 5.1. A campaign of 14,196 error injections was performed over 30 days. The 26 ROS topics cover virtually all the output topics, including localization, perception, planning and control in Autoware; the only topics/nodes that are not covered are those that are specific to the simulator itself (e.g., UI, saving data, visualization).

5.2.2 Error Propagation Analysis

Goal. We analyze the results of the fault injection campaign to understand how different forms of faults in different nodes are propagated to the output of the AV software. In particular, we have three goals. First, we aim to identify, for each injected error, whether it is propagated to the end of the ROS graph and, thus, corrupts the actuator commands. Second, in case an error is masked, i.e., invisible to the actuator, we aim to locate where the error is masked. Finally, we aim to identify the fault masking mechanism used to mask that error.

Note that corruption in the actuator commands *might* not actually cause a safety issue, mainly because the closed-loop control in the software stack would very likely

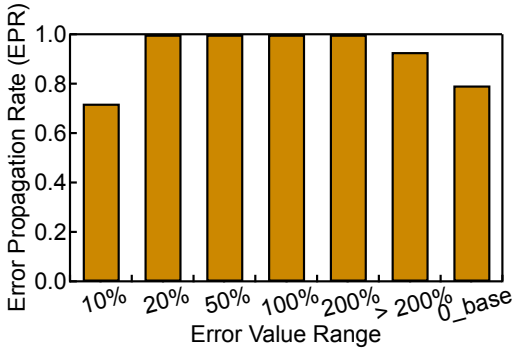


Fig. 5.2: Error propagation rate when injecting error into node *twist_gate* which does not have inherent masking.

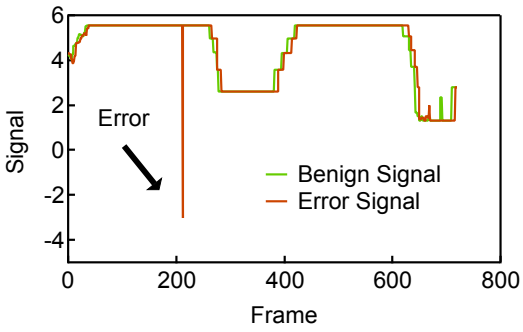


Fig. 5.3: One signal in the instructions to the actuators has been affected by the error.

mitigate a transient command error in the current frame and re-align the vehicle back to the correct trajectory. In this sense, what is reported here is a conservative analysis of AV safety.

Establishing ground truth. To analyze whether and how errors are propagated, we must first obtain the ground-truth, i.e., fault-free results. To that end, we use the same input scenario used in fault injection to drive Autoware and record the output of *each ROS node* under analysis. To accommodate natural run-time variance, the same run is repeated 5,000 times to capture a distribution for each node's output.

Analyzing results. For each fault injection run, if the AV software stack provides no error masking mechanism, the output of the ROS graph, i.e., the actuator commands, will necessarily be corrupted. In contrast, if the actuator commands are uncorrupted (according to some metric), some form of error masking must have taken place between the node where the fault is injected and the output node. Our goal in this section is to identify the masking node. Next section describes the actual masking mechanisms we identified.

We first define that an ROS node is deemed to be corrupted by the fault, i.e., the error is propagated to this node, if the node output is "out of distribution", which is empirically defined as lying outside the bounds of the fault-free range by over three

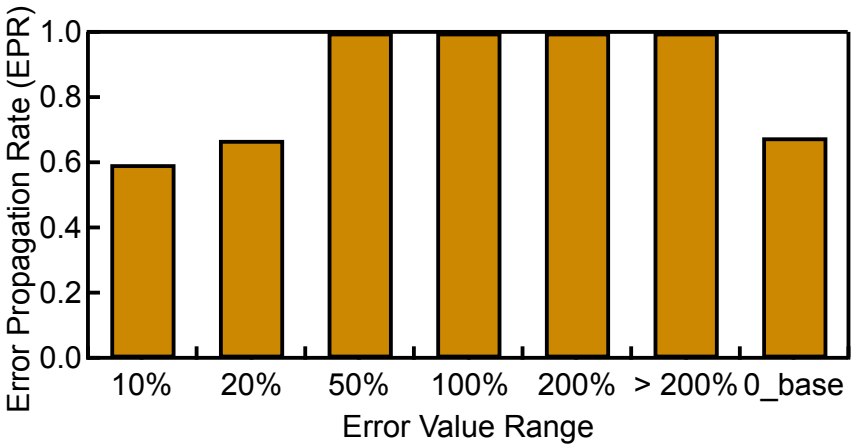
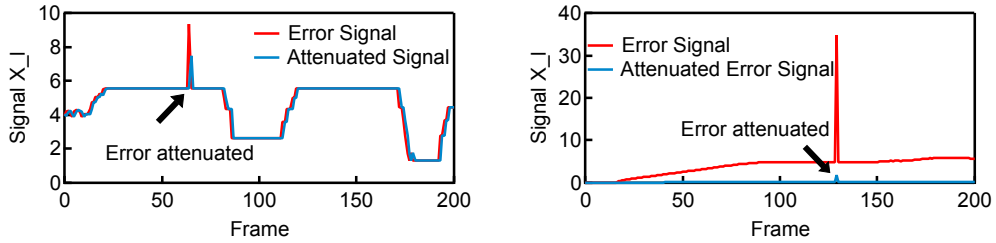


Fig. 5.4: Error propagation rate when inject error into node *twist_filter* which use low-pass filter to attenuate errors.

times of the mean value. This criterion is similar to what is used in prior work (Nitsch et al., 2021).

We analyze each fault injection run in a *backward* fashion, starting from the output node of the ROS graph and check if the fault propagates to this node. If not, the error injected must have been masked somewhere before the node. We then check whether the parent nodes are corrupted. We repeat this process until we reach the node where the fault is injected. This process help us to precisely locate the node, if any, that masks the injected fault.

Once we identify a fault-masking node, we then identify the actual masking mechanism in the node. This is done in a semi-automated way. We statically instrument the code to monitor how each input variable is used in the code. We then re-run the fault injection to capture the statement that masks the error. We then manually examine the code to understand how the error is actually masked. We discuss our findings next.



(a) Example of an error attenuated by the low-pass filter.

(b) Example of an error attenuated by the integral computation.

Fig. 5.5: Examples of signals with error attenuation mechanism such as low-pass filter and integral computation.

5.2.3 Masking Mechanisms

We classify the masking pattern exist in Autoware into four different categories: No Masking (NM), Attenuation (A), Unconditional Masking (UM), and Conditional Masking (CM). We characterize how often errors propagate to the actuator command using Error Propagation Rate (EPR). Tbl. 5.1 shows the EPR of all the nodes we inject.

No masking. Some nodes have no inherent masking mechanisms. Fig. 5.2 shows the EPR of the *twist_gate* node, where the x-axis shows the amplitude of error injection. For example, "10%" on the x-axis means the value after error injection is with in the range of 90% to 110% of the original value. "0_base" means the original value is 0.

Almost all the injected errors are propagated to the output because of a lack of inherent masking in *twist_gate*. The overall EPR is 82.1%. Fig. 5.3 shows an example, where an error injected into *twist_gate* causes drastic changes to the actuator commands.

Attenuation by low-pass filter. Attenuation mechanisms exist commonly in the source code of Autoware. Low-pass filter is a traditional way of filtering out high-frequency signals. Autoware uses a low-pass filter at the end part of the entire compute graph to smooth the output signals and avoid sudden changes. The carefully-designed low-pass filter will degrade the signal with a sudden change in its output, which is effective to errors with low amplitude. Fig. 5.4 shows the error propagation rate when

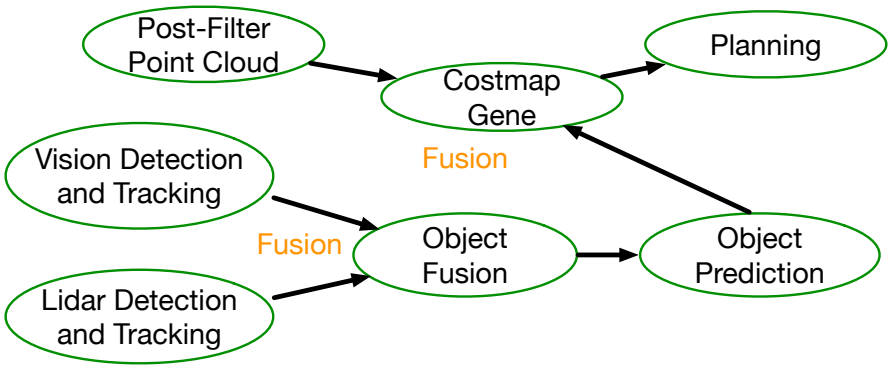
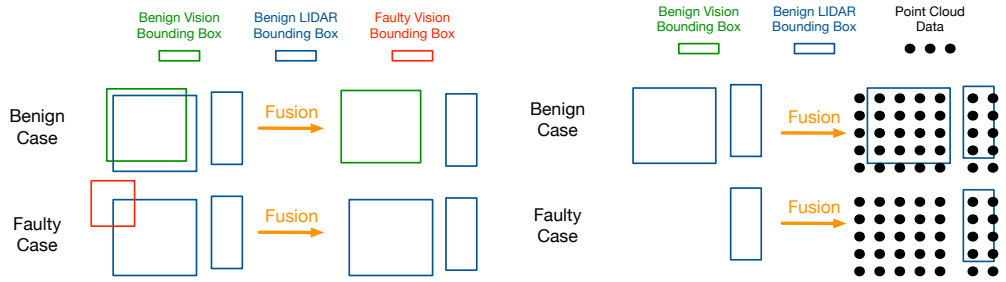


Fig. 5.6: Two types of fusion happen in the perception pipeline of Autoware. The first happen between vision and LIDAR perception, the second happen when creating a map for planning module.

the low-pass filter is applied. The EPR is significantly lower (59.7% in "10%" error value range, 67.0% in "20%" error value range) when the error amplitude is low but remains 100% when the error value range increase. Fig. 5.5a shows an example of the use of the low-pass filter. The error amplitude is significantly reduced from 62.1% to 35.4%.

Attenuation by integral computation. Another typical way of attenuation in Autoware is integral computation. For example, in the localization module, specifically the *ndt_matching* node, the new estimation of pose and localization is calculated by an addition of the previous pose and the difference on distance and pose. The difference is calculated using the velocity estimation and interval on time. An error occurs on the velocity estimation node is largely attenuated as the time step is very small. Fig. 5.15 shows an example where the error is significant on the estimated velocity but attenuated to a small value afterwards.

Unconditional masking in sensor input preprocessing. We find there exist unconditional masking inside AV software which complete mask the errors happen in certain nodes. Unconditional masking technique exists in sensor input preprocessing node. Sensor input preprocessing nodes edit sensor input and sometimes remove redun-



(a) Vision detection and tracking results are fused with corresponding LIDAR branch. Mask the wrong results of vision branch.

(b) The fusion results of vision and LIDAR branch is further fused with the raw point cloud data.

Fig. 5.7: Detail procedure of two fusion processes.

dant parts. For example, raw point cloud data from LIDAR sensor will be first filtered the points representing the ground and then used in point cloud object detection node. Such preprocessing nodes will generate massive sensor data that is naturally robust to errors. We inject errors into a LIDAR point cloud preprocessing node. The point cloud filter will filter out the points that represent the ground. The errors we inject manipulate the coordinates of the points after filtering. We change up to 80 points and monitor the output of the node that consume the faulty point cloud data.

We find that with the increase of faulty points injected into the point could filter node, all the outputs of the consumer node are not impacted with a relative standard deviation (RSD) of 0.002, -0.018 and 0.09 respectively.

Tbl. 5.2 shows the effect of injecting errors into the LIDAR point cloud preprocessing nodes. For each time, we inject an error that changes the coordinates of a certain number of points with an upper bond of 80 (2.9% and 6.7% of total output point cloud respectively). We find that all of these errors are masked and the final output to the actuators are not influenced.

Unconditional masking through multi-sensor fusion. Another unconditional masking technique utilized in AV software is multi-sensor fusion. AV software usually uses more than one source of sensor input during perception tasks such as detection

and tracking. Both LIDAR and camera will capture the objects' information around the vehicle and usually the multi-sensor information will be fused together for higher accuracy and robustness (Gao et al., 2018a; Du et al., 2017).

We find such a process is utilized in Autoware and illustrated in Fig. 5.6. The output object sequences from the vision branch and the LIDAR branch are first fused through a fusion node and used in the prediction module. The results of the prediction module will be fused again with the raw point cloud data in the costmap generation node afterwards. The costmap generation node will use the results of the second fusion to guide the vehicle to search for a path through the obstacle objects. Two continuous fusion techniques ensure that errors in the related perception nodes can be tolerated.

Fig. 5.7a illustrate the first fusion — between vision detection and tracking with LIDAR detection and tracking. The fusion algorithm first check whether the bounding boxes captured by both branches agree on the object label, position, and area. If so, as in the benign case, the fusion algorithm will keep the vision branch's bounding box and those LIDAR bounding boxes that do not have a match. When the vision branch is faulty, the fusion algorithm can not find matching bounding boxes, and the vision bounding box will be *discarded*. Thus, the faulty vision bounding box is masked.

Fig. 5.7b illustrates the second fusion case. The perception module will produce a sequence of bounding boxes after detection, tracking and prediction. The bounding boxes will be fused with the raw point cloud data to produce a map for the planning module. In the second fusion, a union operation will be performed on the bounding boxes and point cloud raw data to create the map. Thus, even if the faulty bounding boxes are not masked by the first fusion process, the error will be masked in the second process.

The EPR results verify the unconditional masking through multi-sensor fusion. We inject three kinds of errors, which are label errors, bounding box location errors and size errors, into all the nodes related to vision perception and LIDAR perception branch. All of them are masked and will not influence the output signal to the actuators.

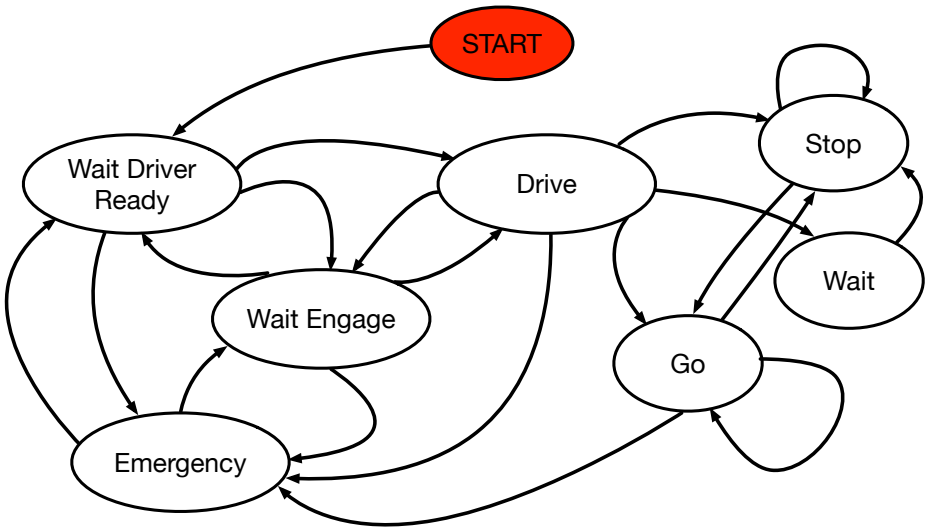


Fig. 5.8: Motion state machine used in Autoware.

Conditional masking in state machine. Error masking under conditions also exists in Autoware. One of the most common examples is the conditional masking in the state machine of Autoware. Autoware manages the vehicle status with three complicated state machines: mission, motion and behavior state machine. Fig. 5.8 shows an illustration of the motion state machine. Eight different states with various transformation conditions consist of the motion state machine and provide enormous conditional masking patterns. For example, if in one frame the current state is *Wait Engage*, all the errors happen on the signals related to the *Go, Stop* and *Wait* states are masked as they are irrelevant to the *Wait Engage* state. Thus, all the nodes that produce these signals will not influence the final outputs as well.

Conditional masking in *If* statements. If statements create conditional masking patterns. Lst. 5.1 shows the snippet of the *velocity_set* node, which has has eight input signals and three output signals. With a driving scenario where both `condition1` and `condition2` are not satisfied, `obstacle_point` and `stopline_point` will not be read — even if they are corrupted.

To confirm this, Fig. 5.9 shows the EPRs when faults are injected to three nodes that generate inputs to the *velocity_set* node. In our particular runs, `condition1`

and `condition2` are not satisfied; thus, faults in *pose_relay* and *velocity_relay* are naturally masked by *velocity_set*. However, `condition3` holds in certain runs; thus, the EPR under *astar_avoid* is not zero. Fig. 5.10 shows the execution traces. Fig. 5.10a compares the values of `obstacle_point`, an output of *velocity_set*, when a fault is injected to *astar_avoid*. Fig. 5.10b shows the value of `final_point`, another output of *velocity_set*, when a fault is injected into *pose_relay*. One can see that faults in the former are masked by *velocity_set* but faults in the latter are not.

Listing 5.1: Error masked by conditional if-statements

```
1  Inputs: signal: current_pose, current_vel,
2        points_no_ground,
3        detection_range, cross_walk,
4        decelerate_obstacle_point,
5        obstacle_point, stopline_point, safety_point;
6        function: f,g, mp1, mp2, mp3
7  Outputs: obstacle_point, stopline_point, final_point
8
9  function velocity_set(){
10   Bool condition1, condition2, condition3;
11   condition1 = f(detection_range, cross_walk);
12   condition2 = f(detection_range, cross_walk);
13   condition3 = g(decelerate_obstacle_point);
14
15   if (condition1 || condition2)
16   {
17      obstacle_point = mp1(current_pose, current_vel,
           points_no_ground);
18      stopline_point = mp2(current_pose, current_vel,
           points_no_ground);
19   }
```

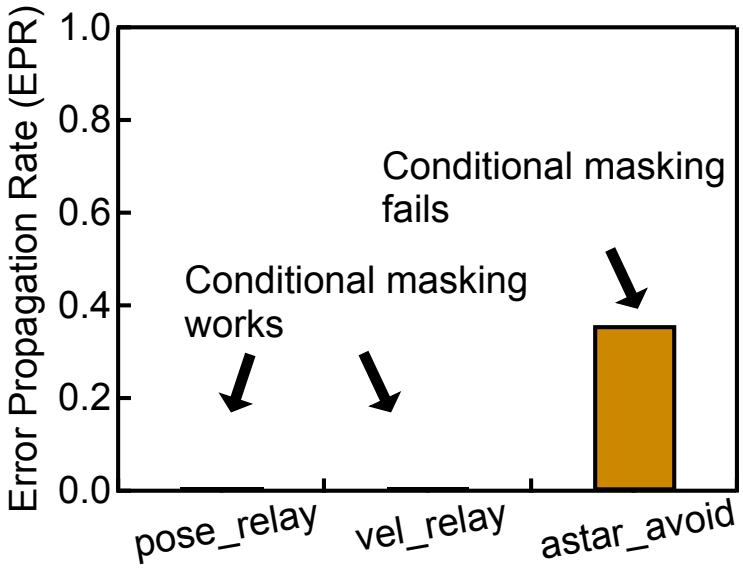
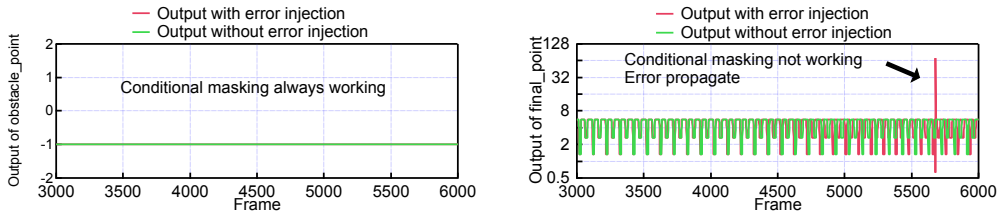


Fig. 5.9: EPRs when faults are injected to three producers of *velocity_set*.

```
20  else
21  {
22      if (condition3)
23      {
24          final_point = mp3(safety_point);
25      }
26  }
27  return obstacle_point, stopline_point, final_point
28  }
```

Summary. The understanding of different fault masking/attenuation mechanisms allow us to classify the fault tolerance level of an Autoware node. This is shown in Tbl. 5.3. Among them, only two nodes (twist_gate and decision_maker) have no masking patterns. Both of them are at the end part of the Autoware graph and directly contributed to the output of the AV software. Two nodes have attenuation mechanisms. All the nodes in the perception pipeline are with unconditional masking mechanisms using two fusion operations. The rest nodes have conditional masking mechanisms.



(a) Faults in *pose_relay* are masked.

(b) Faults in *astar_avoid* are not masked.

Fig. 5.10: Outputs of the *velocity_set* node when faults are injected into its two producers *pose_relay* and *astar_avoid*.

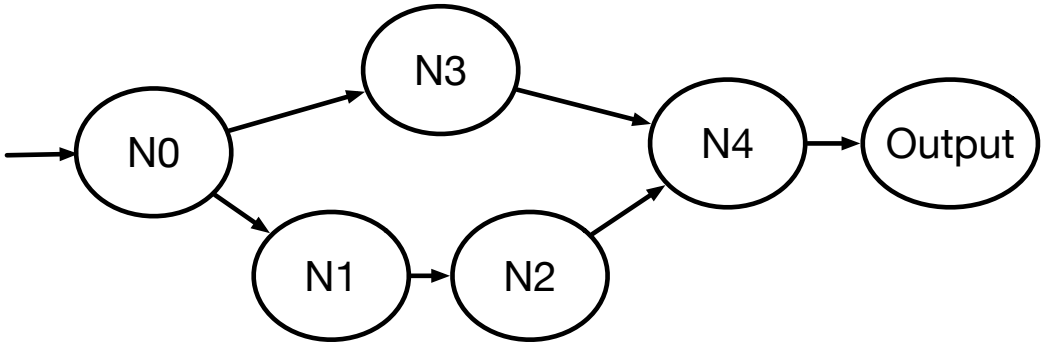


Fig. 5.11: A simple graph to illustrate how the dynamic FTL of a node is calculated.

We show the EPR in the last column. Not surprisingly, nodes without any masking mechanisms have the highest propagation rate, both are above 80%. Attenuation with a low pass filter can prevent a small number of errors from propagating and reduce the EPR to 69.2%. Integration shows a much stronger masking effect, only 17.8% of the errors finally propagate to the end and most of them are with large error values. Nodes with unconditional masking inherently are robust and have 0% EPR for all seven nodes. Nodes with conditional masking have various EPR, ranging from 0% to 36.3%.

5.2.4 Calculating Fault Tolerance Level

After each node is assigned with its inherent masking pattern, we can derive a node's fault tolerance level (FTL), which indicates whether/how a node's output errors can be masked/attenuated before reaching the actuator. Let us use a simple graph in Fig. 5.11,

where we aim to calculate the FTL of node $N0$, to describe the intuition behind our algorithm.

First notice that $N0$'s sub-graph has two separate paths: $N1 \rightarrow N2 \rightarrow N4$ and $N3 \rightarrow N4$. That is, there are two paths for $N0$'s output error, if any, to be propagated to the output. Therefore, the FTL of $N0$ is the weaker of the two paths. Given a path, say, $N3 \rightarrow N4$, the error masking mechanisms of all nodes on the path are applied sequentially; therefore, the FTL of a path is the strongest FTL of all the nodes on the path. For instance, if $N3$ has inherent attenuation and $N4$ has inherent unconditional masking, the FTL of the path $N3 \rightarrow N4$ is unconditional masking, because the attenuation effort is "overwritten" by the unconditional masking. The overwriting behavior of unconditional masking, a stronger masking, takes place regardless of whether it occurs before or after attenuation, a weaker masking.

Formally, we define a partial order "weaker than", denoted $\prec$, between three masking mechanisms:

$$\mathsf{NM} \prec \mathsf{A} \prec \mathsf{UM}$$

Note that a conditional masking node, by definition, will be dynamically resolved to either no masking or unconditional masking depending on whether the node is triggered.

Given the intuition above, the FTL of $N0$ in Fig. 5.11, $F(N0)$, is calculated by the following equation:

$$F(N0) = min(max(F(N1), F(N2), F(N4)), \\ max(F(N3), F(N4))) \tag{5.1}$$

Notice how the equation is inherently iteratively defined: calculating $F(N0)$ requires first calculating $F(N1)$, $F(N2)$, $F(N3)$, and $F(N4)$. This suggests that to calculate the FTL of a node one must start from the *output* node of the graph. In practices,

we reverse all the directions in the ROS DAG, start from the output node to perform a breath-first search, and calculate the FTL of all the nodes in a single traversal pass. The FTL of the output node is NM.

Tbl. 5.3 shows the FTL of all the nodes. We make two observations from Tbl. 5.3. First, a node's FTL might be different from its inherent masking mechanism. This is because the FTL of a node, say P, depends on all *other* nodes between P and output. Second, the FTL of certain nodes vary at run time if there exists a CM node on the path to the output.

5.3 The BRAUM Protection System

Based on the fault tolerance characterization, we propose a dynamic protection system based on analyzing the node vulnerability in the Autoware software stack. We first discuss a baseline protection mechanism (Chapter 5.3.1), followed by our protection scheme that reduces the protection overhead with little accuracy impact (Chapter 5.3.2).

5.3.1 Baseline Protection Mechanism

We first describe a baseline protection strategy, which is representative of those commonly found in literature (Hsiao et al., 2021) and provides strong protections at the cost of high overhead. Our work, however, does not fundamentally depend on this baseline scheme. Fig. 5.12 illustrate the baseline protection system described here.

The protection strategy operates at a node granularity. The basic idea is to monitor the output of a node and detect if the output is an outlier based on certain distribution. If an outlier is detected, we then re-execute the node (i.e., temporal redundancy). We call this "output outlier detection and re-execution" (OODR). In particular, we use a Gaussian-based Anomaly Detector (GAD). We maintain a mean value and a standard deviation σ in a fixed size window of 10 frames.

When the value of the output is N standard deviations away from the mean value, the recovery will be triggered. Critically, N is configurable based on the nature of the nodes. For example, we could use a larger N if a node's FTL is A, since slight fluctuation in the output is likely attenuated by the subsequent nodes. N would be higher or lower for nodes classified as NM and UM, respectively. The particular recovery scheme we consider is to re-execute the current node.

In addition to detecting the output outliers (and re-executing an abnormal node), we also detecting input outliers. In particular, we use the same GAD to detect any outlier in the inputs. If an outlier is detected, we replace the input with the an input value that is within the N sigma distance. We call this "input outlier detection and resetting" (IODR).

Sharp readers might wonder why such an input outlier detection and resetting is necessary: wouldn't protecting every node's output effectively protect every node's input? The reason is three-fold. First, it is possible that not all the nodes' outputs are protected, especially when a node's implementation is provided by a third-party library or when protecting a node is simply to costly (e.g., re-execution takes too much time). Second, re-execution does not fundamentally mitigate faults introduce by software bugs or adversarial attacks, for which input outlier detection and resetting is known to be effective (Hsiao et al., 2021). Finally, a fault-free output could be corrupted during data transmission.

5.3.2 Our Protection System

They key insight behind our protection system is that if errors/faults in node's output can be inherently masked or attenuated later, one can relax the protection strength of the node and thus reduce the protection overhead. Algo. 2 shows the overall protection system.

At run time once a node finishes its execution, we first calculate the dynamic FTL

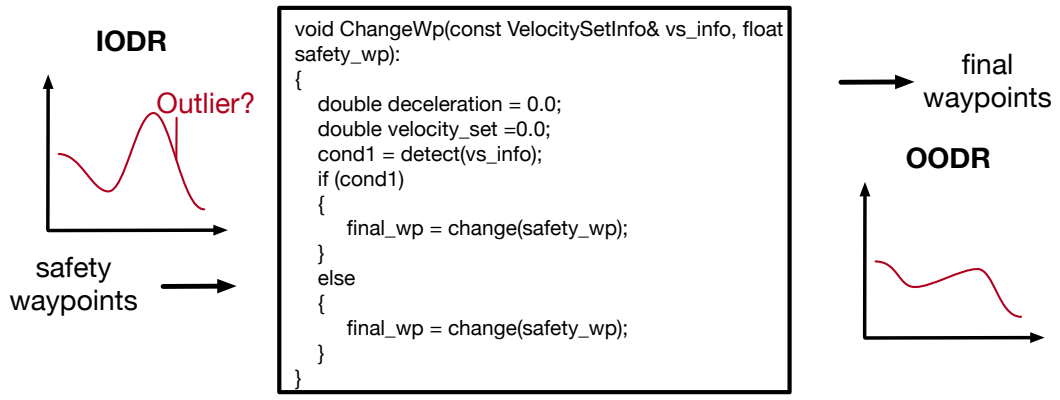


Fig. 5.12: Baseline protection system in BRAUM.

of the node using the algorithm described in Chapter 5.2.4 (an example is shown in Equ. 5.1). Recall that this step must be executed dynamically for each frame, since a node's fault tolerance mechanism depends on how a CM is resolved at run time.

Calculating the FTL of a node requires us to resolve all the yet-to-be executed CM nodes in the sub-graph of the current node. Thus, we must predict how each downstream CM is resolved. For simplicity, we use a last-value predictor, i.e., using the resolution of CM in the last frame as the prediction of the current frame. This is inspired by recent work that shows that autonomous machine states have temporal consistency (Li et al., 2022b), where sudden state changes are rare.

If a node's dynamic FTL is NM, the outlier detection and temporal re-execution is triggered as usual. However, if a node's FTL is stronger than NM, we could potentially elide the re-execution. Specifically, if a node's dynamic FTL is UM, we can skip outlier detection and temporal re-execution altogether, since any output error is expected to be masked down the line. If a node's FTL is A, we relax the outlier detection threshold (use a larger N in Chapter 5.3.1) in that slight change in output could be attenuated later in the execution. Relaxing the outlier detection threshold could reduce the frequency of node re-execution, improving performance.

Handling Mis-predictions. Just like a mis-prediction in a processor must be dealt with to avoid incorrect pipeline execution, the mis-prediction of a CM node's resolution

Algorithm 2: BRAUM protection system.

Input: Current node T; $T(\cdot)$ represents the execution of the node; fault masking mechanism of each node in ROS graph; output outlier detection threshold N; slack in outlier detection k.

Resolve all the CM nodes in the graph;

$FTL_T \leftarrow$ Calculate the FTL of the current node T;

if FTL_T *is UM* **then**
  $T(\cdot)$;
end

if FTL_T *is NM* **then**
  Run input outlier detection and resetting with threshold N;
  $T(\cdot)$;
  Run output outlier detection and re-execution with threshold N;
end

if FTL_T *is A* **then**
  Run input outlier detection and resetting with threshold N;
  $T(\cdot)$;
  Run output outlier detection and re-execution with threshold $N + k$;
end

if *fault masking mechanism of* T *is CM* **then**
  if T*'s resolution target is mis-predicted* **then**
    Re-evaluate the FTL for all the nodes that are before T in the ROS graph;
    Re-execute the current frame from the first nodes whose actual FTLs are weaker than previous predicted;
  end
end

must be taken care of as well. In particular, mis-predicting the resolution of a CM node P could alter the FTL of a node that P depends on. For instance in Fig. 5.11, if $N2$ is a

CM node that should have been resolved to NM but, instead, is predicted as UM, both $N0$ and $N1$'s FTL should be re-calculated, resulting in different protection schemes for both nodes.

Our observation is that before entering a CM node we know exactly whether it would be resolved as UM and NM, from which we know whether we have mis-predicted the resolution target of this CM node. Upon a mis-prediction, we will recalculate the FTLs of all the nodes whose FTLs depend on ths CM node. If the actual FTL of a node, say P, is *weaker* than the predicted FTL, that means the protection strategy applied to P should have been *stronger*.

To deal with mis-predictions, we identify the first node in the entire ROS graph whose actual FTLs are *weaker* than what were previously predicted and re-execute the current frame from there. Note that while mis-prediction does increase the frame latency, it does *not* affect a vehicle's behavior because the output of mis-predicted execution would not have reached the vehicle actuator yet when the mis-prediction is detected.

5.4 Evaluation

We first describe the evaluation setup (Chapter 5.4.1) and demonstrate the effectiveness and overhead (Chapter 5.4.2).

5.4.1 Evaluation Setup

Nodes with protection. As a proof of concept, we pick four representative ROS nodes to implement our protection system, assuming that faults take place in only those nodes.

- *twist_gate*, whose FTL is NM.
- *twist_filter*, whose FTL is A.

- *detection_lidar_detector*, whose FTL is UM.
- *velocity_set*, whose FTL is either NM or UM, since one of its subsequent nodes has conditional masking.

Baselines. We compare with four different baselines.

- **Base**: the vanilla AV software without any fault protection.
- **IODR+OODR**: a system that performs both input outlier detection and resetting (IODR) and output outlier detection and re-execution (OODR).
- **IODR**: a system with only IODR.
- **OODR**: a system with only OODR.

Implementation. Our protection system is implemented at the software level by modifying the Autoware source code after the analysis is performed. The source code of the four vulnerable nodes is enhanced with our protection mechanism and Autoware is then re-built from source. For evaluation, we inject the exact same errors as in Chapter 5.2 to test the effectiveness of BRAUM protector.

5.4.2 Protection Results and Overhead

Faults in *twist_gate*. Fig. 5.13 compares the EPR before and after the protection scheme is applied. BRAUM reduces the error propagation rate from 80.6% down to 8.3%. Fig. 5.15a shows a concrete example of how the protection scheme masks the error in *twist_gate*. We inject a significant error at frame 220 and BRAUM successfully detects the error and recovers the correct value. Both **IODR** and **OODR** are unable to achieve similar protection result. They reduce the EPR to 35.7% and 40%, respectively. **IODR+OODR** achieves same protection effectiveness compared to BRAUM.

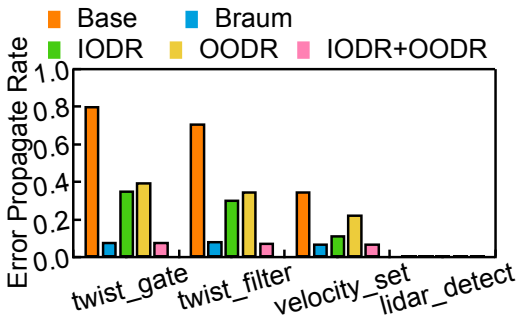


Fig. 5.13: Error propagation rate largely reduced when BRAUM protection is applied.

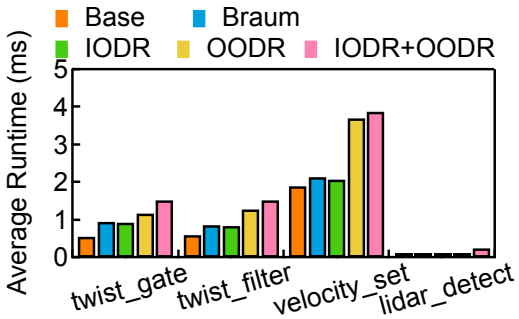
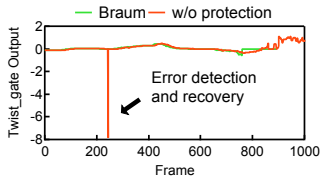
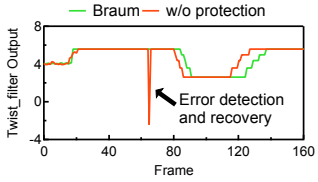


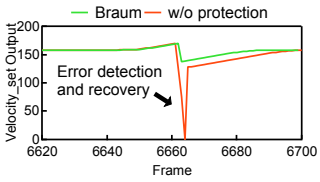
Fig. 5.14: Runtime overhead is minimum when BRAUM protection is applied.



(a) Example of an error detected and recovered in twist_gate node.



(b) Example of an error detected and recovered in twist_filter node.



(c) Example of an error detected and recovered in velocity_set node.

Fig. 5.15: Concrete examples of how the BRAUM protection works.

The reason EPR is not further reduced is that in certain cases, the input signal does have a sudden change, yet the protector treats it as an outlier and replaces it with the average value in the previous window. An error created by the protector thus will propagate and result in a non-perfect protector. For example, in *twist_gate*, 97.3% of the protection failure cases are caused by false positives. The data is 94.5% for *twist_filter*.

Faults in *twist_filter*. The FTL of *twist_filter* is A (Tbl. 5.3), for which we empirically loose the output outlier detection threshold to 6 $sigma$. Loosening the detection threshold can help reduce the frequency of re-execute the code, thus saving protection overhead. BRAUM protection reduces the EPR of *twist_filter* by 87.6%, **IODR** is able to reduce the EPR by 56.6% and the result for **OODR** is 50.1%. Fig. 5.15a shows an example of how the protection scheme works. **IODR+OODR** achieves slight better

EPR (8.1%) compared to BRAUM.

Faults in *velocity_set*. Fig. 5.13 shows the error propagation rate after the protection is applied on *velocity_set*. The EPR reduce from 36.3% to 7.6%. We also find that the simple predictor we implemented has a very high accuracy. The mis-prediction rate is only 2.7%. Fig. 5.15c shows a concrete example of error mitigated in *velocity_set*.

Faults in *detection_lidar_detector*. Although we do not perform any protection on *detection_lidar_detector*, the EPRs are all 0 due to unconditional masking patterns.

Protection on unseen errors. We evaluate our protection method on a set of unseen errors with the same scenarios. The unseen errors has the same amount compared to the original evaluation and within the same range of error amplitude.

For the new error set, the EPR (lower the better) in the 4 nodes are 8.2%, 7.2%, 8.3%, and 0%, similar to the current results reported in the paper (8.3%, 8.8%, 7.6% and 0%), indicating our method is still effective in unseen errors. We also perform another evaluation on the protection method by using a different scenario with a different error set compared to the analysis phase. As an example, if faults take places in the node *twist_gate*, our protection method reduces the EPR to 9.1%, similar to the 8.3% results we currently have.

Protection overhead. BRAUM protector is lightweight and brings minimum overhead to the AV systems. We show the average runtime comparison between the baseline and after applying BRAUM protector in Fig. 5.14. For *twist_gate* and *twist_filter*, the overhead is relatively higher (50.1% in *twist_gate* and 43.3% in *twist_filter*). This is because those two nodes are extremely lightweight that do not perform any complicated computation, both have an average runtime of less than 0.6ms. The overhead reduce to 11.2% in *velocity_set*. 0% overhead is introduced on *detection_lidar_detector*.

As compared to **IODR**, BRAUM has an average 2.5% higher runtime overhead as the use of output protection. However, the runtime is saved by 47.2% compared to **OODR**, as the always re-execution adds enormous overhead. **IODR+OODR** has the

highest protection protection overhead, BRAUM saves runtime by 55% compared to **IODR+OODR**.

As a comparison with existing protection methods, we compare with DeepFense (Rouhani et al., 2018) and Dual-core Lock Step (DCLS) (Peña-Fernández et al., 2019). DeepFense utilizes redundancy to protect perception modules. BRAUM achieves the same protection accuracy with 93.75% less run-time overhead. DCLS is a hardware mechanism to detect and recover from hardware transient faults. BRAUM reduces the error protection rate from 35.2% to 7.9%. DCLS requires two times hardware area overhead, whereas BRAUM requires none.

5.5 Related Work

Error injection into software. Simulating the errors that can possibly happen in software has been studied by prior works. Such errors include soft errors (Hsiao et al., 2021; Porpodas, 2019), adversarial attacks (Madry et al., 2017; Yuan et al., 2019) and software bugs (Garcia et al., 2020). Most of these works try to relate errors they simulate with a metric such as mission success rate and quality of service (QoS) to demonstrate the errors injected do create reliability issues. We, however, instead of only caring about how many errors finally propagate to the end with EPR metric, also try to understand what types of fault tolerance mechanisms successfully mask the errors at the software level.

Masking of the errors. Error masking has been studied by various previous works. For example, the masking of soft errors can happen at circuit level (Baze et al., 2000; Lin et al., 2010), architectural level (Mukherjee et al., 2003; Borodin and Juurlink, 2010) and single program level (Fang et al., 2016; Sridharan and Kaeli, 2008). We try to go one step further, understanding the inherent fault tolerance of the entire AV software stack, which is consisted of tens of different programs. We assume that the errors

have already passed all the masking mechanisms in the bottom level of the computing stack such as the circuit and architectural level and result in a wrong output of the program. Under such a basis, we further study if specific operations (i.e., integration) or interactions from multiple programs (i.e., fusion) can stop the errors from propagating.

Protecting AV software. To counter errors happen in AV software, two categories of protection methods have been proposed. The first one is utilizing modular redundancy, both temporal (Kim, 1999; Rivers, 1998; Oplinger and Lam, 2002) and spatial (Lyons and Vanderkulk, 1962; Kastensmidt et al., 2005). The second category is to detect anomaly outputs and recover (Hsiao et al., 2021). Most methods are agnostic to all the nodes in AV software and bring heavy overhead. We propose selective protection, which spends limited resource on the most vulnerable nodes such as the ones with no fault tolerance mechanisms inherently.

Table 5.1: List of ROS nodes this paper analyzes.

Module	**ROS node**	**EPR**
Sensor Preprocessing	ray_ground_filter	0%
	voxel_grid_filter	0%
Localization	can_odometry	0%
	ndt_matching	23.4%
	pose_relay	8.7%
	vel_relay	2.4%
Perception	vision_darknet_detect	0%
	vision_beyond_track	0%
	detection_lidar_detector	0%
	detection_lidar_tracker	0%
	range_vision_fusion	0%
	naive_motion_predict	0%
	costmap_generator	0%
Planning and Control	astar_avoid	0%
	velocity_set	36.3%
	decision_maker	100%
	pure_pursuit	17.8%
	lane_stop	0%
	lane_rule	26.9%
	twist_filter	69.2%
	twist_gate	80.6%
	lane_select	0%
	waypoint_planner	0.68%

Table 5.2: All the errors injected into the nodes that preprocess LIDAR point cloud data are masked.

ROS node	**# of input points**	**# of faulty points injected**	**EPR**
Ray_ground_filter	[1682,4019]	[0,80]	0%
Voxel_grid_filter	[915,1349]	[0,80]	0%

Table 5.3: Fault tolerance level (FTL) of all the Autoware ROS nodes under evaluation.

Node	Masking mechanism	FTL
twist_gate	NM	NM
decision_maker	NM	NM
twist_filter	A	A
pure_pursuit	A	A
vision_darknet_detect	NM	UM
vision_beyond_track	NM	UM
detection_lidar_detector	NM	UM
detection_lidar_tracker	NM	UM
range_vision_fusion	NM	UM
naive_motion_predict	NM	UM
costmap_generator	NM	NM/UM/A
ray_ground_filter	UM	UM
voxel_grid_filter	UM	UM
astar_avoid	CM	NM/UM
velocity_set	CM	NM/UM
lane_stop	CM	NM/UM
lane_rule	CM	NM/UM
waypoint_planner	CM	NM/UM
ndt_matching	CM	NM/UM/A
can_odometry	CM	NM/UM/A
pose_relay	CM	NM/UM/A
vel_relay	CM	NM/UM/A
lane_select	CM	NM/UM/A

6 Heterogeneous Split-Lock Architecture for Safe Autonomous Machine Software

As the number of transistors per chip continues to grow and operating voltages decrease, the likelihood of soft errors is increasing (Chatterjee; Bhuva, 2018). Soft errors can lead to silent data corruption (SDC), which poses a significant reliability threat in autonomous machine systems (Lotfi et al., 2019). Additionally, autonomous machines are synonymous with safety-critical applications such as self-driving vehicles, making reliability a first-class concern.

Spatial redundancy is an effective method for addressing SDC caused by soft errors. Previous research demonstrates that incorporating additional hardware redundancy is beneficial for improving system reliability (Iturbe et al., 2016b, 2019; LaFrieda et al., 2007). However, implementing spatial redundancy at the architectural level is extremely inefficient. Lock-step CPU designs, which involve executing identical software on two cores and comparing their results to ensure functional correctness, serve as an effective method for mitigating soft errors in space applications. Yet, applying this lock-step CPU design in autonomous machines presents two challenges. First, such designs significantly reduce the available computational resources in autonomous machine systems, potentially affecting their functionality, especially when there is a tight silicon

budget for most autonomous machine systems and keep increasing the number of cores is impossible. Second, traditional lock-step systems employ inefficient error correction methods. Typical approaches involve re-executing the entire software or performing a system-wide restart. Both methods are unsuitable for autonomous machines since several seconds of system downtime could compromise the safety of these devices.

In the previous chapter, BRAUM indicates that different algorithms in a typical autonomous machine software stack have different inherent fault tolerance and thus the protection on them could be selectively. BRAUM validates the idea with a simple protection mechanism.

In this chapter, following the findings in BRAUM, I propose a practical system that can mitigate resource contention encountered in lock-step CPU implementations and enable efficient error correction. Our core insight is that tasks within autonomous machine software exhibit varying degrees of fault tolerance against errors. In traditional system designs, reliable tasks and vulnerable tasks share the same hardware resources. Consequently, I introduce a multi-domain system design comprising a lock domain and a split domain. This configuration allows more reliable tasks to run in the split domain to enhance performance, while vulnerable nodes operate in the lock domain to ensure reliability.

I offer flexible macros that users can utilize to annotate their code, indicating the inherent fault tolerance of each task. By implementing instances of these macros, I ensure that the operating system scheduler assigns different tasks to their desirable hardware domains, optimizing both performance and reliability.

When an error occurs in the lock domain, I provide a specific routine for error correction. I find that due to the nature of almost every autonomous machine software, different tasks are connected through a producer-consumer message passing mechanism, and each task is inherently idempotent. This means that an error occurring within a task will only impact the entire system when it publishes messages to its consumers. This insight enables us to save the register values at the beginning of each task and only

re-execute the corresponding task to correct the error when detected by the hardware. Our proposed method, KINDRED, does not require a system-wide reboot, which would otherwise result in intolerable overhead.

I emulate our approach in a real autonomous machine system to assess the performance degradation. I implement KINDRED in an open-source full-stack autonomous vehicle software called Autoware. Our approach successfully improves the system performance by an average of 11.0% compared to a system with only lock-step CPUs. These performance improvements translate to a 3.1% higher success rate across various driving scenarios.

In summary, KINDRED presents a practical system for reliability issues caused by soft errors in autonomous machine systems while maintaining performance. In summary, I have three contributions in this work. First, I describe a novel multi-domain system designed to mitigate soft errors in lock-step CPUs while maintaining performance. Second, my approach offers flexible APIs that enable users to strategically allocate vulnerable tasks to the lock domain and performance-critical tasks to the split domain. Third, by harnessing the idempotent nature of programs, I facilitate lightweight error recovery through context saving and re-execution. Notably, our error correction methodology can be seamlessly integrated into real systems without necessitating hardware modifications.

6.1 Background

We describe the fault model first and relevant background on Lock-step CPU design in this section.

Silent data corruption caused by random bit flips. Random bit flips are a common type of hardware error that exist in current computing systems (Nicolaidis, 2010). A range of factors can cause random bit flips in a computing system, including electrical noise, cosmic rays, and abnormal temperature fluctuations (Baumann, 2005a;

Lantz, 1996; Li et al., 2016). In CPU design, random bit flips can occur at any position, such as cache, registers, Arithmetic Logic Unit (ALU), and control path. The soft error rate (SER) can easily exceed 50,000 failures in time (FIT) (Baumann, 2005a).

Our fault model focuses on single bit flips in hardware, which represent the majority of cases in random bit flips (Ayatolahi et al., 2013; Nicolescu et al., 2004; Elliott et al., 2013). When a random bit flip occurs in the hardware, various consequences can arise in the software. Many random bit flips can be masked at the micro-architecture level (Sridharan and Kaeli, 2010), architecture level (Mukherjee et al., 2003), and software level (Sridharan and Kaeli, 2008; Fang et al., 2016). When errors propagate to the process, three different outcomes may occur: the process can crash, hang, or have a silent data corruption (SDC) in the output (Hsiao et al., 2021). Crashing or hanging can be easily detected and resolved by restarting the process. However, an SDC in the output can cause severe consequences for the system behavior, such as converting a brake command into acceleration in autonomous driving applications (Gan et al., 2022).

Lock-step redundant CPU systems. To combat random bit flips, researchers have proposed introducing redundancy in system design. Lock-step system design involves executing the same software on multiple hardware components to ensure the correctness of the software. The concept of lock-step can be applied to CPUs, GPUs, NPUs, and other components in a system-on-a-chip. This paper primarily focuses on lock-step CPU design for two reasons. First, although GPUs have been utilized in the perception modules for self-driving vehicles, the majority of autonomous machine software, such as those running on mobile robots and drones, use CPUs as their hardware. Second, autonomous machine software contains a large portion of code responsible for control and planning, which are tasks that are naturally not suitable for running on accelerators.

Lock-step CPU design detects and recovers from random bit flips by pairing multiple CPU cores and comparing the results among them (Iturbe et al., 2019; de Oliveira et al., 2018; LaFrieda et al., 2007; Aggarwal et al., 2007). Each CPU core within the pair runs the exact same software, and their results are compared at every cycle for

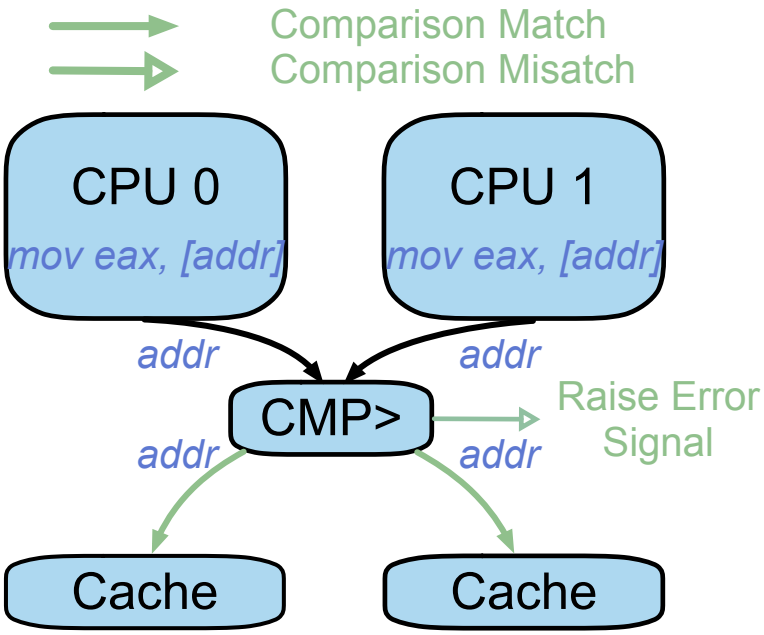


Fig. 6.1: Dual-core lock-step CPU design.

correctness.

We illustrate a typical lock-step CPU design in Fig. 6.1. Both cores in the lock-step CPU are running the same process. Thus, the same instructions will be executed in the CPU pipeline. In order to avoid introducing extra stages of comparison into the CPU pipeline, which will result in significantly high overhead, we compare the output ports of the CPU cores.

In the example shown in Fig. 6.1, a *mov* instruction is in the pipeline waiting for data to be loaded from memory. If there are no errors, both CPU cores will fetch data from the same address. Thus, we compare the addresses of the data loading. If the two cores request data from different addresses, we can infer that an error has occurred in one of the CPU cores. If an error is detected, we raise an error signal and attempt to recover from the error. On the other hand, if the addresses match in both cores, the execution is not influenced by any soft errors, and it can proceed. Comparing the output ports of CPU cores introduces the least overhead to the current CPU design.

Unlike error correction in a dual-core lock-step system, error detection and correc-

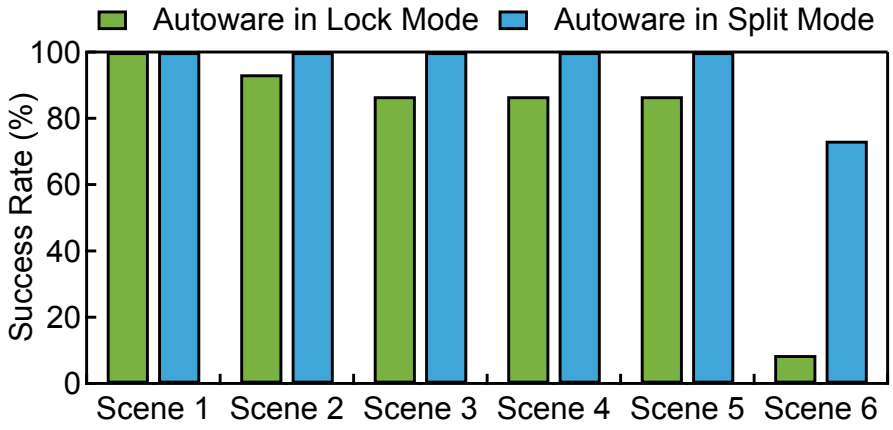


Fig. 6.2: Success rate comparison between lock mode and split mode.

tion in a triple-core lock-step system are addressed in hardware. This is because an odd number of cores can help determine which core is faulty, enabling direct recovery from the other two benign cores without the need for a checkpoint. The extra hardware includes a majority voter, a checker and a resynchronization logic (Iturbe et al., 2019).

6.2 Motivation

To motivate the roadblocks preventing the practical use of lock-step CPU design, we first show in this section how the mission success rate of an autonomous machine system is significantly impacted by using a lock-step CPU design (Chapter 6.2.1), and the root cause is higher average latency caused by resource contention. We then describe they key motivation of our work, i.e., the different levels of inherent robustness in an autonomous machine software stack (Chapter 6.2.2).

6.2.1 Lock-step CPUs Degrade Mission Success Rate

We run the open-source self-driving software Autoware (Kato et al., 2018) and use the Intel CARLA simulator (Dosovitskiy et al., 2017) to provide environment simulations.

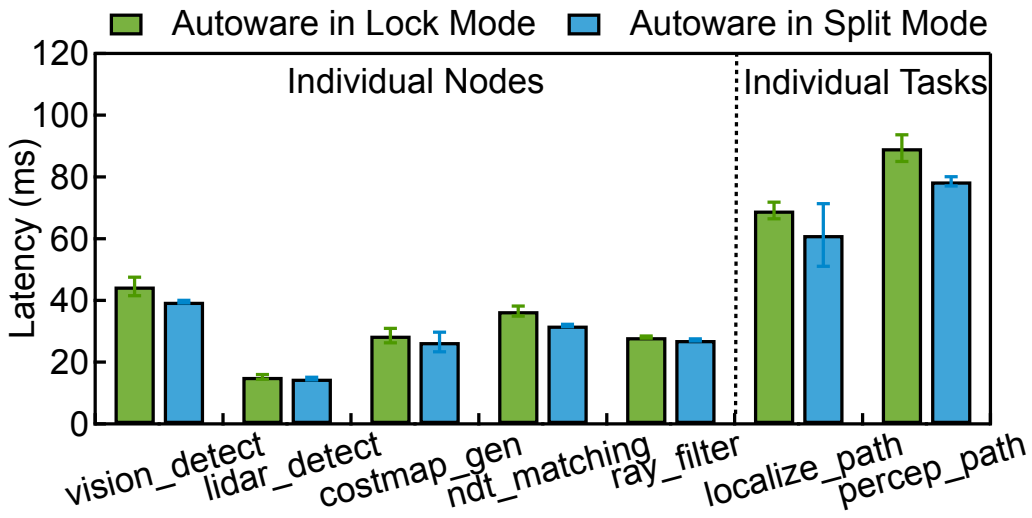


Fig. 6.3: Latency comparison on node different nodes and pipelines.

The setup runs on a server with an eight-core Intel Xeon W-2123 CPU and an Nvidia Quadro RTX 4000 GPU. We instrument the source code to profile the performance.

Workload. We create various workloads in CARLA and run an ego vehicle with Autoware as its self-driving software inside the environments. Different workloads include cruising, following other vehicles, emergency stops, and more.

Settings. We employ a simple yet effective method to faithfully mimic the lock-step CPU design. There are two reasons why we do not characterize performance in real systems. First, most desktop-level and server CPUs do not support lock-step mode, as they do not have exceptionally high reliability constraints. Second, at the time of this work's submission, the only commercialized development board—the Nvidia Orin, which features lock-step ARM A78AE CPUs—does not provide well-documented information on lock-step mode, and it also does not adequately support the autonomous machine software and corresponding simulator.

For the lock-step mode, we run Autoware on two cores. We bind all the processes of Autoware to core 0 and core 1. For the split mode, we run Autoware on four cores, with all the processes bound from core 0 to core 3. We try our best to eliminate the influence of other processes by binding processes like the simulation environment to

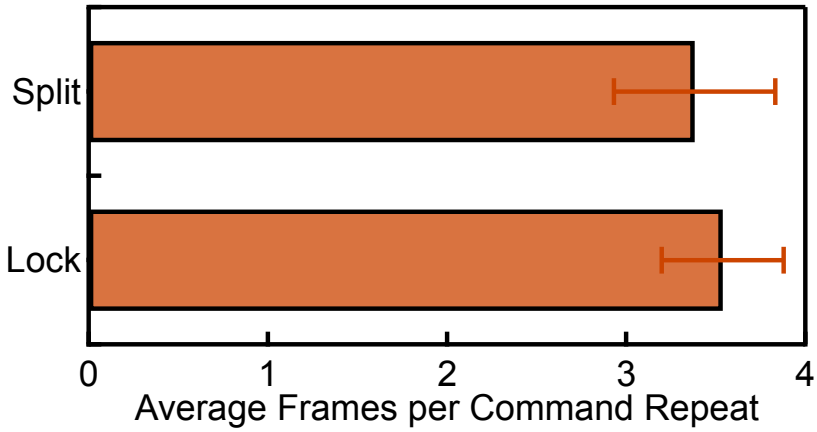


Fig. 6.4: Comparison on average frames repeat per command.

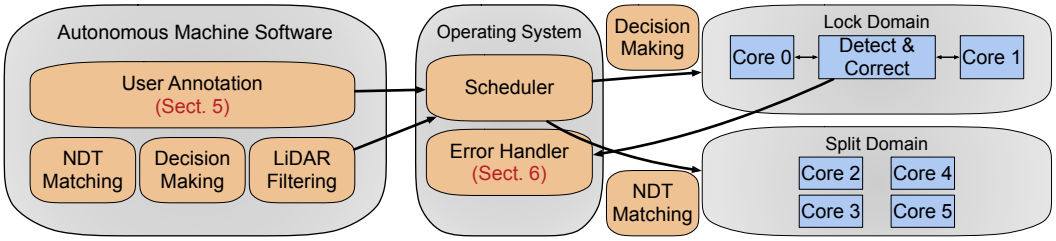


Fig. 6.5: KINDRED System Overview.

other specific cores.

Success Rate. One of the most critical metrics for autonomous machines is whether they can successfully complete their tasks. To assess the impact of lock-step mode on success rate, we create six different scenarios in the simulator and run an ego vehicle equipped with Autoware inside it. We set up the starting point and end point for the ego vehicle, as well as the trajectory of the participants, such as other vehicles and pedestrians. The complexity of the different scenarios increases from scenario one to scenario six. A higher complexity entails longer traveling distances and increased interaction with other vehicles and pedestrians.

Fig. 6.2 shows the success rate of the ego vehicle when running in lock-step mode and split mode. The results are averaged from thirty runs in every scenario. Running in lock-step mode significantly reduces the success rate. In easier scenarios, such as scenarios one to five, the difference is less obvious. However, in the most complicated

scenario, running in lock-step mode has a 64.7% lower success rate compared to running in split mode. More failures, such as colliding with other vehicles and veering onto the sidewalk, occur.

Root-cause Analysis. The performance degradation in lock-step mode is primarily due to reduced computational resources in the system. Half of the CPU cores are used as backup cores to guarantee reliability. This results in resource contention, leading to a slowdown in Autoware. We observe the average CPU utilization increases from 71.8% to 82.9%.

We further analyze the detailed latency data to demonstrate that resource contention indeed leads to longer latency in autonomous machine software pipelines. By profiling the latency at various stages of the pipeline, we can effectively identify bottlenecks and understand how resource contention impacts the overall performance of the autonomous machine software.

We monitor nodes in various pipelines, such as vision detection and LiDAR detection in the perception pipeline, and nodes like ndt_matching and ray_ground_filter in the localization pipeline. Additionally, we track the pipeline latency for both perception and localization in order to understand the impact of resource contention on these critical components of the autonomous machine software.

We compare the performance of lock-step mode (referred to as lock mode later) and split mode in Fig. 6.3. We first show the runtime latency of individual nodes and task pipelines. As expected, running Autoware in lock mode significantly degrades runtime latency. When operating in lock mode, all nodes experience longer runtime latency, with an average slowdown of 9.9%. This slowdown accumulates, resulting in increased latency in the task pipeline. The average latency for perception increases by 13.9%, and for localization, by 12.9%.

In addition to average latency, another metric that reflects the reduced computation resources is the average repetition time for each command issued by the vehicle. Autoware, like many other autonomous driving software systems, operates under real-time

constraints. Control algorithms that directly generate commands for the vehicle actuator must operate at a minimum frequency of 30 Hz. This ensures that the autonomous machine can effectively and safely respond to dynamic situations in its environment and maintain stable operation.

When nodes preceding control algorithms cannot meet the 30 Hz frequency, the controller node will simply repeat or duplicate its previous output, allowing a new actuator command to be generated. Consequently, the frame repeat rate serves as a valuable metric that offers insight into how the system's latency is affected by resource constraints.

We find that the average frames repeated per command does indeed increase. As shown in Fig. 6.4, in split mode, every command is repeated on average 3.38 frames. This number increases by 4.7% to 3.54 frames when Autoware is running in lock mode.

Overall, the motivational data shows that using lockstep considerably hurts the vehicle's mission success rate, which, in turn is caused by the longer pipeline execution latency as a result of reduced computation resources. As a result, designers either have to commit much more computation resources (chip area) to guarantee latency, or simply opt out of lockstep execution; neigher is desirable.

6.2.2 Diversity of Inherent Fault Tolerance of Autonomous Machine Software

A key observation that KINDRED relies on is the inherent fault tolerance of the autonomous machine software. The software stack of an autonomous machine consists of tens of algorithms, which possess different abilities to tolerate faults, depending on their inherent logic and the data flow of the entire software. Recent work performs a thorough fault injection campaign and classifies autonomous machine algorithms into four different levels depending on how robust they are against faults (Gan et al., 2022): unconditional masking (UM), conditional masking (CM), attenuation (A), and no masking

(NM).

Unconditional Masking. A node with fault tolerance level UM means that when an SDC happens on it, the error will not propagate through the entire software stack and reflect on the final output. For example, if an autonomous machine software stack fuses perception results from multiple sensors. Errors on one of the sensors can be mitigated.

Conditional Masking. A node with fault tolerance level CM means that when an SDC happens on it, the error sometimes propagate to the output of the AV software. For example, AV software manages multiple different state machines to manage its mission, motion and behavior. If the vehicle is in "Go" state, errors happen on the conditions related to "Go" state will be propagated. However, the errors happen on conditions related to a "Wait" state will be masked.

Attenuation. A node with fault tolerance level A means that when an SDC happens on it, the error will not be masked but will be attenuated and propagate to the output of the AV software. A low-pass filter is a typical way of reducing the amplitude of an error.

No Masking. The nodes with lowest fault tolerance level is classified as NM. SDCs happen on these nodes will directly influence the output of the AV software, which are the control commands on steering wheel, gas and brake pedals.

6.3 KINDRED Overview

Main Idea. Our main intuition behind this work is to intelligently allocate the protection budget by considering the robustness/vulnerabilities of different algorithms in the autonomous machine software stack. In particular, algorithms that are inherently robust against soft errors (e.g., those that can naturally mask soft error-induced SDCs) can execute in the split mode (i.e., without the protection of lockstep execution), whereas algorithms that are vulnerable to soft errors (e.g., those that can only attenuate or have no ability to mask SDCs) should be protected through lockstep execution.

Architecture. To that end, KINDRED introduces a heterogeneous architecture that consists of both a split domain and a lock domain. CPUs in the split domain execute completely independently, and CPUs in the lock domain execute strictly in a lock-step manner.

Fig. 6.5 shows the overview of KINDRED system. We divide the CPU cores in the system into two different domains, the lock domain and the split domain. The lock-step design can be either a dual-core lock-step where two cores are bound together or a triple-core lock-step. Each core still has private level-1 instruction and data cache, as well a private level-2 cache.

The split-lock architecture is reminiscent of the classic asymmetric multiprocessors architectures that consist of power-hungry high-performance (big) cores and power-efficienct, low-performance (little) cores (Kumar et al., 2003, 2004). Just like how a big-little system can balance power and performance by intelligently scheduling tasks onto different cores (Zhu and Reddi, 2013), the key of our split-lock heterogeneous architecture is also to decide when and how to schedule an algorithm (in the autonomous machine software stack) to which domain.

User Annotations. In order to decide how to schedule algorithms to the two domains, we rely on software developers to annotate the vulnerability of different nodes as shown in Fig. 6.5. This is because the software designer has the best understanding of the vulnerability of each node. For example, in most autonomous machine software, the control node that outputs commands to the vehicle actuators tends to implement a low-pass filter to remove sudden changes in the front-end. While this might be difficult for an offline compiler analysis to figure out, the developer can easily annotate the code to indicate the level of robustness as attenuation (A) (Chapter 6.2.2).

The implementations of the annotation APIs, provided in a user-space library, decide how to schedule an algorithm onto the different cores in the two domains. We elaborate the annotation APIs and its run-time implementations in Chapter 6.4.

Hardware-Software Collaborative Detection and Recover. Traditional lock-

step system either resort to fully hardware support for error detection or use software-only solutions. For example, LaFrieda et al. propose leveraging a multi-stage hardware compression circuits to compress register files every fixed interval and compare all the register file contents (LaFrieda et al., 2007). Software-only solutions usually treat an error as an interrupt and apply software restart inside the corresponding interrupt service routine (ISR).

We provide a hardware-software collaborative mechanism as shown in Fig. 6.5, where error detection is done in hardware and error recovery is done in software. In particular, the lock-domain CPUs naturally have dedicated logic that compare the output wires of each core in the domain every cycle and raises a signal to the OS when the results of the cores mismatch. The OS, in the kernel space, runs a mis-match handler, or error handler, which is triggered when the mis-match signal is received. The handler is responsible for recovering the system from the mis-matches while minimizing the performance overhead.

While the hardware error detection logic is readily available on commercial CPUs with lockstep execution capabilities (Chapter 6.1), the design of the mis-match handler and its interactions with applications and hardware is novel to this work, which we describe in Chapter 6.5.

Listing 6.1: Example Autoware Code

```
#include "twist_filter/twist_filter_node.h"
#include "scheduling_hints.h"

int main(int argc, char** argv)
{
  SCHED_HINT_ATTENUATION_PROCESS;
  ros::init(argc, argv, "twist_filter");
  twist_filter_node::TwistFilterNode node;
  ros::spin();
```

```
10   return 0;
11 }
```

6.4 Robustness Annotations

We rely on software designers to annotate their code for us to schedule. We provide different macros for the software designers to identify the reliability of each nodes. To be specific, software designers can use macros to indicate the vulneraebility level of the process in their code. We show an example of twist_filter in Code 6.1.

Listing 6.2: Macro Definition Header File

```
1  #ifndef SCHEDULING_HINTS_H
2  #define SCHEDULING_HINTS_H
3
4  // Function declarations
5  void scheduling_hint_attenuation_process();
6  void scheduling_nomask_process();
7  void scheduling_condmask_process();
8  void scheduling_uncondmask_process();
9
10 // Macro definitions
11 #define SCHED_HINT_NOMASK_PROCESS
       scheduling_hint_attenuation_process()
12 #define SCHED_HINT_ATTENUATION_PROCESS
       scheduling_nomask_process()
13 #define SCHED_HINT_CONDMASK_PROCESS
       scheduling_condmask_process()
14 #define SCHED_HINT_UNCONDMASK_PROCESS
       scheduling_uncondmask_process()
```

```
15 #endif
```

The SCHED_HINT_ATTENUATION_PROCESS macro indicates that twist_filter node is with vulnerability level to be attenuation. We provide corresponding header file contains the macro definitions in Code 6.2.

We then implement the corresponding schedule functions using sched_affinity system call to schedule the corresponding process on different domains. Assuming the lock domain contains two logic cores, core zero and core one and split domain contains cores between two to five. We will set the affinity for the process to only core zero and core one if it should be scheduled on the lock domain. Otherwise, we will set the affinity for the process to core zero to core five, as the tasks can run on the split domain does not have to run on the split domain, and can also be scheduled on the lock domain.

6.5 Error Detection and Correction

Error detection (in the lock domain) is done by hardware. We claim no novelty here. In particular, when the results of the cores in the lock domain do not match, the hardware logic will stall both cores and raise a signal to the Generic Interrupt Controller (GIC). The GIC then relays the signal to the OS (Abdelrazek, 2006).

Our contribution lies in how we efficiently recover from the error, upon receiving the mis-match signal, by leveraging characteristics unique to autonomous machine software. We elaborate on the unique task-level idempotency inside AV software in Chapter 6.5.1. We then describe how exactly we recover from an error in Chapter 6.5.2.

6.5.1 Exploiting Task Idempotency for Lightweight Recovery

In existing lockstep systems, when a mis-match takes place, the fauty process/program must be restarted and, if the entire system has no other spare cores (i.e., all the cores participate in lockstep execution), the entire system must reboot (to reset the faulty cores). Restarting the program or rebooting the system is clearly unacceptable in an autonomous systems: it leads to significant latency increase or, in the worst case, discrupts the functionality of the vehicle.

The goal of our error recovery mechanism is thus to introduce little latency overhead to the software stack without any disruption, i.e., restarting the entire program or rebooting the entire system is unacceptable. Our key idea is to leverage the idempotency of tasks in autonomous machines. We will first describe the inherent idempotency in the software stack and describe how to leverage this for lightweight recovery.

Task-level Idempotency. In most autonomous machine software, different tasks operate in independent processes that communicate with each other through messages. Fig. 6.6 shows a simple example illustrating this programming model, where "pub" denotes publishing a message to subsequent nodes and "sub" denotes subscribing a message from preceding nodes. In the example node 0 is the producer of node 1 and node 2, and node 2 is the producer of node 3. Node 0 receives input from its producer task (the `sub` block in Fig. 6.6), performs computations on them (the `Code` or `Idempotent Code` block in Fig. 6.6), and then publishes the output to its consumer task (the `pub` block in Fig. 6.6).

Crucially, nodes communicate with each other strictly through messages. No global variables that live through function calls exist. A consumer node can not start its execution until it receives the message as its input from the producer node. Therefore, each node is idempotent (De Kruijf and Sankaralingam, 2013), meaning that however many times a node's code is executed, the output messages will be the same as long as the

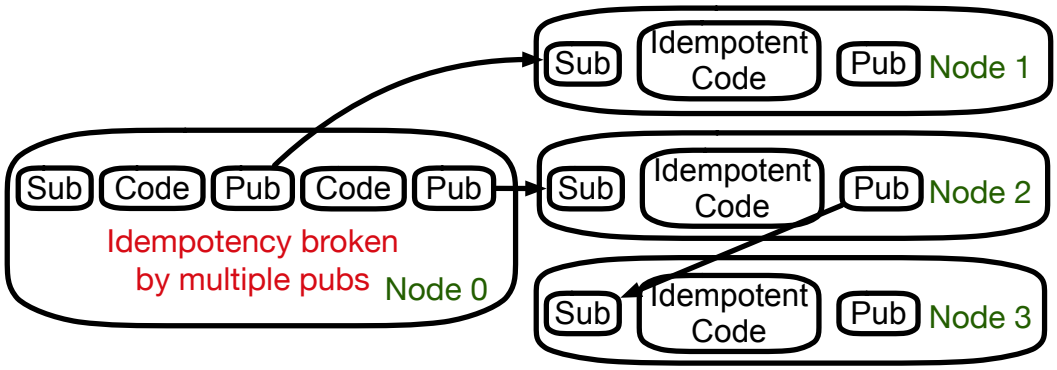


Fig. 6.6: Message sharing through multiple nodes in autonomous machine systems.

input messages are the same.

Listing 6.3: Task-level Idempotency

```
#include <pcl/registration/ndt.h>



static void imu_callback(const sensor_msgs::Imu::Ptr&
    input))
{
    // Handle input data;
    imu = *input;
    // Code with idempotency;
    save_registers();
    predict_pose_imu = imu_calc();
    // Publish output data;
    predict_pose_imu.publish(predict_pose_imu);
}

static void odom_callback(const
    nav_msgs::Odometry::ConstPtr& input)
{
    // Handle input data;
```

```
18      odom = *input;
19      // Code with idempotency;
20      save_registers();
21      odom_res = odom_calc(input->header.stamp);
22      // Publish output data;
23      odom_pub.publish(odom_res);
24  }
25
26
27  int main(int argc, char** argv)
28  {
29      SCHED_HINT_CONDMASK_PROCESS;
30      ...
31      ros::Subscriber imu_sub;
32      ros::Subscriber odom_sub;
33      ...
34      return 0;
35  }
```

Lightweight Recovery. The fact that each algorithm (node in Autoware) in autonomous machine software stack is idempotent provides a unique opportunity for lightweight error recovery without restarting the application: whenever a SDC is detected by the lockstep hardware, we simply have to re-execute only the current node that is being executed when the soft error occurs. Re-execution here means restoring the architectural states when the node is first entered and executing the entire code in that node using the restored states. Since the node is necessarily idempotent, re-execution will generate the same output message and, thus, guaranteeing the same functionality to the rest of the pipeline.

In order to restore the architectural states, we must first save them. KINDRED

achieves so by instrumenting the original autonomous machine software code with a `save_registers()` function call at the beginning of each node, as shown in line 9 and line 20 of Code 6.3. The saved architectural states include both general-purpose registers (`RAX`, `RBX`, `RIP`, `R8` - `R15`, etc.) and floating point registers (`XMM0` - `XMM7`, etc.). Each process saves its states into a dedicated place in the data segment of kernel space so architectural states from different processes will not pollute each other.

Our approach has two main benefits. Firstly, it ensures that none of the potential errors will be written to the consumer nodes or propagate to the system as it has not reachd the message publish phase. Secondly, if two tasks execute serially, we can avoid extra overhead by not re-executing from the first task when an error occurs in the second one.

Listing 6.4: Multi Consumer Example

```
1  static void points_callback(const sensor_msgs::Imu::Ptr&
       input))
2  {
3     // Handle input data;
4     ......
5     // Code with idempotency;
6     save_registers();
7     predict_pose_points = points_calc(filtered_scan_ptr);
8     // Publish output data;
9     predict_pose.publish(predict_pose_points);
10    // Another piece of code with idempotency;
11    ndt_pose = ndt_calc(filtered_scan_ptr);
12    // Publish output data to another consumer;
13    ndt_pose_pub.publish(ndt_pose);
14 }
15
16 static void points_callback_reimp(const
```

```
    sensor_msgs::Imu::Ptr& input))
{
    // Handle input data;
    ......
    // Code with idempotency;
    save_registers();
    predict_pose_points = points_calc(filtered_scan_ptr);

    // Another piece of code with idempotency;
    ndt_pose = ndt_calc(filtered_scan_ptr);
    // Publish output data at the end;
    predict_pose.publish(predict_pose_points);
    ndt_pose_pub.publish(ndt_pose);
}
```

Multiple Consumers. For this scheme to work, the key requirement is that a node commits its output message only at the very end. Otherwise, it is possible that an message is already made visible to a node's consumer when a soft error-induced SDC takes place, in which case re-executing the entire node would generate a duplicate copy of the message to the consumer node, changing the semantics of the application.

Committing a message in the middle of a node's execution is most common when a node has multiple consumers. As an example, consider Node 0 in Fig. 6.6. We show a concrete example from Autoware in function `points_callback` in Code 6.4.

To address this issue, our idea is to delay all the output message committments to the end of the node. This is done by statically transfroming the code offline. As an example, see `points_callback_reimp` in Code 6.4, which is transformed from `points_callback` by delaying all the message committments.

Hard Real-time Constraints. Re-executing a node necessarily introduces latency overhead, which in the worst case could violate the real-time constraint of a node. For

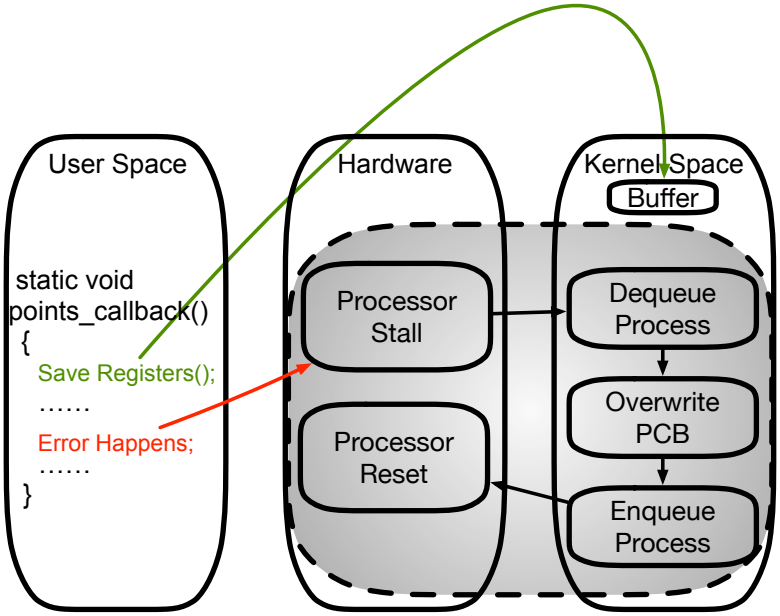


Fig. 6.7: Illustration of error handler in a dual-core lock-step system. The shadow part represents the error handler.

instance, the node that connects to the vehicle and generates control commands need to meet the hard real-time constraint of operating at, e.g., 30 Hz, which must not be violated.

To prevent tasks from missing their deadlines, our approach is to install a timer callback to signal when the deadline is approaching, in which case instead of finishing the re-execution of the node we simply duplicate the most recent message that was previously generated. Re-publishing previous message is a common strategy commonly seen in autonomous machine software to ensure that the system continues to operate without interruption (Bezemer and Broenink, 2015; Kay and Tsouroukdissian, 2015)

6.5.2 Error Handler

We show the error handler in Fig. 6.7. We introduce minimum hardware support to the cores and most recovery steps are done in the software. When the errors are detected, we first stall the processor. We chose to stall the processor in hardware as doing it in

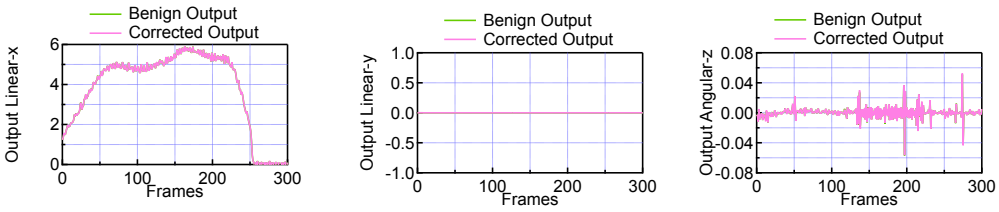
software such as writing to a register, will take hundreds of cycles and the errors will propagate into architectures that are in a clean state, such as Level-1 cache.

After the processor is stalled, a mismatch error signal is sent to the OS to trigger the corresponding error handling code, or error handler. Our proposed error handling mechanism must be executed promptly, in contrast to traditional interrupt handlers which can tolerate delays.

The software portion of the error handler will do three things. First, it will dequeue the processes from the faulty core. This step includes two operations. First, it will save the values architectural registers of the running processes on the faulty core. We will assume the running process is `twist_fillter` in the following text. Notice that the architectural states saved can be faulty as we compare the output ports of the locked CPU cores instead of adding an extra pipeline to compare the architectural state table of two cores. Second, the error handling code will dequeue all the processes in the runqueue of the faulty core.

Next, before it enqueues the processes into the runqueue of other cores, we need to resolve the hazardous values of architectural registers saved during the first step. Because of the dual-core setting, there is no way to determine which core has the correct architectural states. Thus, the error handler needs to recover the architectural states of the running processes to the correct states. It copies from the most recently saved correct values of architectural registers to the process control block of `twist_fillter`. In that case, although `twist_fillter` needs to jump back to a previous location compared to where it stops, the correctness of that process is ensured.

Finally, the error handler will enqueue the dequeued processes in the runqueue of the faulty cores to runqueue of other benign cores. The last thing the error handler does is to write to reset register of the faulty core to reset that core. After the reset, it goes back to the benign state and joins the resource pool so that the OS can schedule processes to it again.



(a) `linear-x` for ndt_matching. (b) `linear-y` for ndt_matching. (c) `angular-z` for ndt_matching

Fig. 6.8: Comparison with the state-of-the-art methods

6.6 Functional Validation

In this section, we demonstrate the error correction method applied for the dual-core system is functionally validated. Since at the time of submission lockstep CPUs, while physically available in hardware, are not accessible to the systems software, our validation and evaluation will necessarily be based on emulating the domain from off-the-shelf (split) CPU cores. We first introduce the experimental methodology in Chapter 6.6.1 and show the validation results in Chapter 6.6.2.

6.6.1 Methodology

Error injection. We emulate the effect of SDCs caused by soft errors by inject errors into randomly selected general-purpose registers or floating point registers by writing arbitrary values into the selected registers. Note that while the microarchitectural behavior of a single soft error is that a single bit in a flop is changed, at the software level multiple architectural registers might be affected and their "corrupted" values could very well be different from simply flipping a bit. Our fault injection faithfully emulates these effects.

Error Recovery. We implement the mis-match handler as a kernel module. After the error is injected, the kernel module sends a SIGSTOP signal to the faulty running process, which emulates the stalling of the fault process (which is normally done by the

lockstep CPUs in hardware). The mis-match handler will then get a handler to the PCB of the faulty process, and overwrite all the architectural registers with the most recently saved values (by `save_registers()` in the application code; see Chapter 6.5.1). Finally, the handler sends a SIGCONT signal to the fault process to resume its execution, emulating the effect of a fault processor being reboot and becoming available afterward.

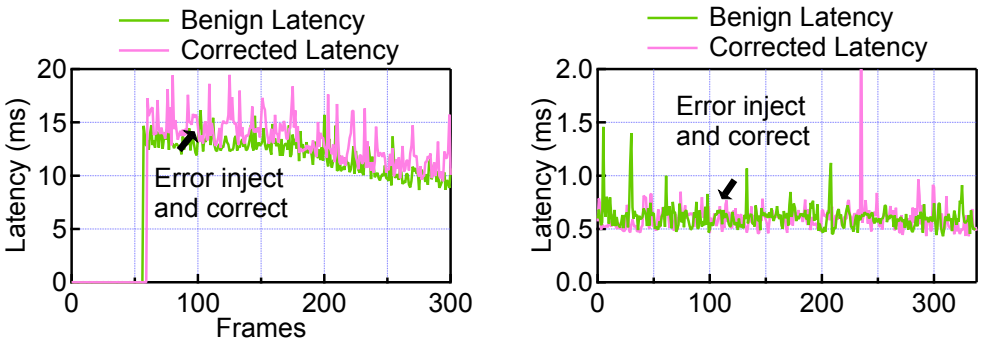
Workloads and Environment. We implement the mis-matched kernel module is implemented in a Linux kernel version 5.14. To verify the effectiveness of the error correction, we created identical scenarios for both benign and faulty runs. We first set up a scenario in CARLA and ran the Autoware software in benign mode. We recorded two things: the output of the specific node or process that we later injected errors into, and the CARLA simulator-related signals, so that we could recreate an identical scenario. This allowed us to compare the output of the software under benign and faulty conditions in the same environment.

We then use the recorded CARLA simulator signals to recreate the exact same scenario. We modify Autoware and load the kernel module to simulate our KINDRED system. During the faulty run, we also record the output of the same node or process. We later compare both outputs to determine if the error we injected actually caused a deviation in the behavior of the vehicle or it has been corrected.

6.6.2 Validation Results

Error correction results. We use the localization node `ndt_matching` as an example to illustrate the correctness of KINDRED. We show the results of the estimate_twist signal, which is the output of `ndt_matching`. We show three output variables `linear-x`, `linear-y` and `angular-z`.

We compare the benign output and the output after error injection and correction in Fig. 6.8. All three variables show the same results between benign experiment and error



(d) Latency comparison on node `ndt_matching`.

(e) Latency comparison on node `voxel_grid_filter`.

Fig. 6.9: Latency comparison between benign experiment and error inject and faulty experiment. Corrected Latency is the latency of the experiment when we inject error and recover from it.

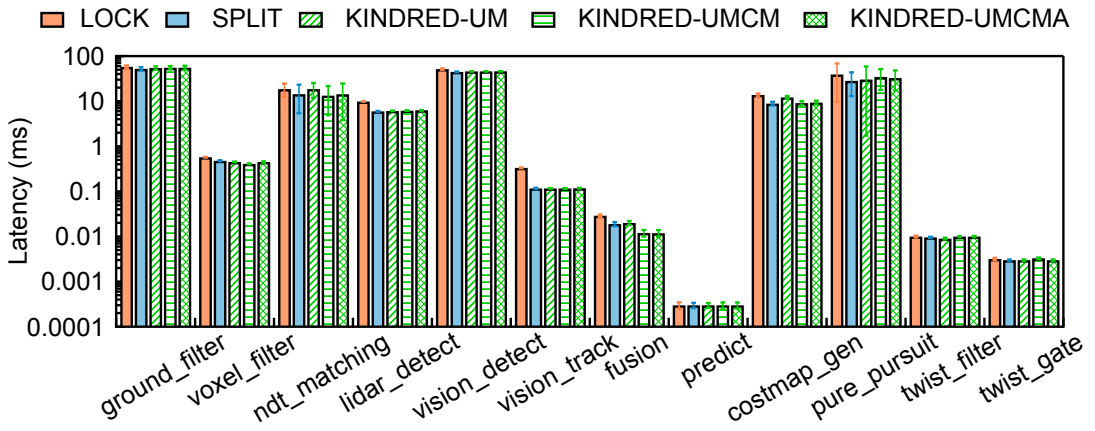
injected and correction experiment. We demonstrate that although with error injected to the architectural registers, our error correction mechanism is able to correct them.

Latency Overhead. We also show the latency overhead we introduce with the error correction. We compare the latency of two nodes during the baseline experiment and the error injection and correction experiment. We show the result in Fig. 6.9.

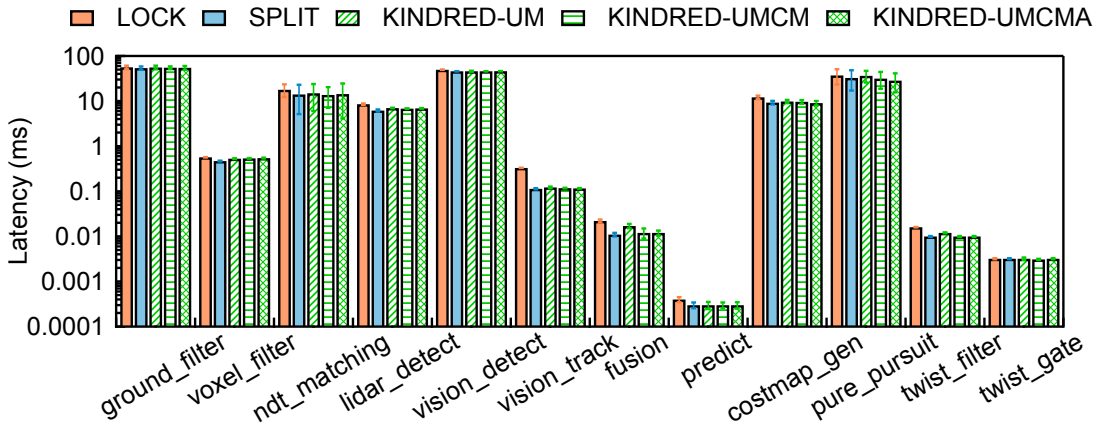
Our error correction methodology does not introduce much overhead. In `ndt_matching` node, KINDRED shows 6.6% higher latency compared to baseline experiment. However, in `voxel_grid_filter`, KINDRED shows 1.8% lower latency.

6.7 Evaluation Setup

We introduce the hardware setup (Chapter 6.7.1) and software infrastructure (Chapter 6.7.2) in this section. We also describe different settings of the KINDRED (Chapter 6.7.3).



(a) Node latency comparison between baselines and KINDRED with a dual-core setting.



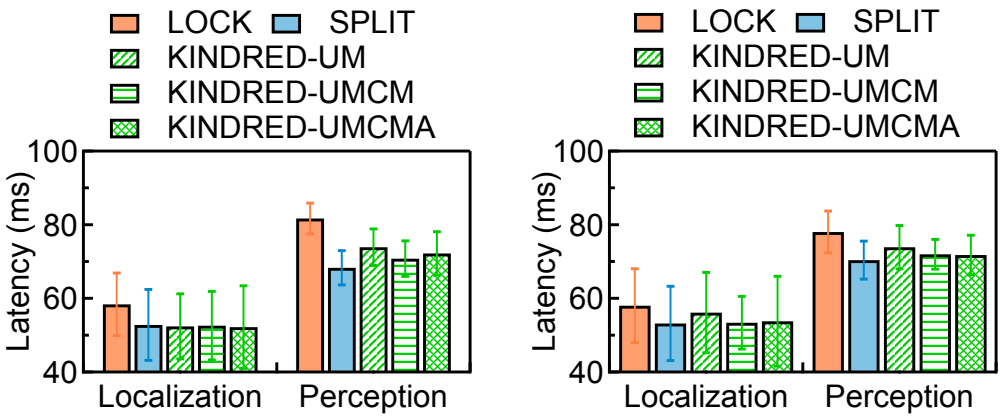
(b) Node latency comparison between baselines and KINDRED with a triple-core setting.

Fig. 6.10: Node latency comparison.

6.7.1 Hardware Setup

We emulate the entire system using existing hardware. The main reason we use emulation instead of simulation is that we want to evaluate the performance of the entire autonomous machine software stack. Running Autoware and CARLA on a cycle-accurate simulator would take an excessively long time to complete due to the complexity and computational overhead involved in accurately simulating every processor cycle.

We develop RTL implementation of the extra hardware required for our project,



(a) Task pipeline latency comparison between baselines and KINDRED with a dual-core setting.

(b) Task pipeline latency comparison between baselines and KINDRED with a triple-core setting.

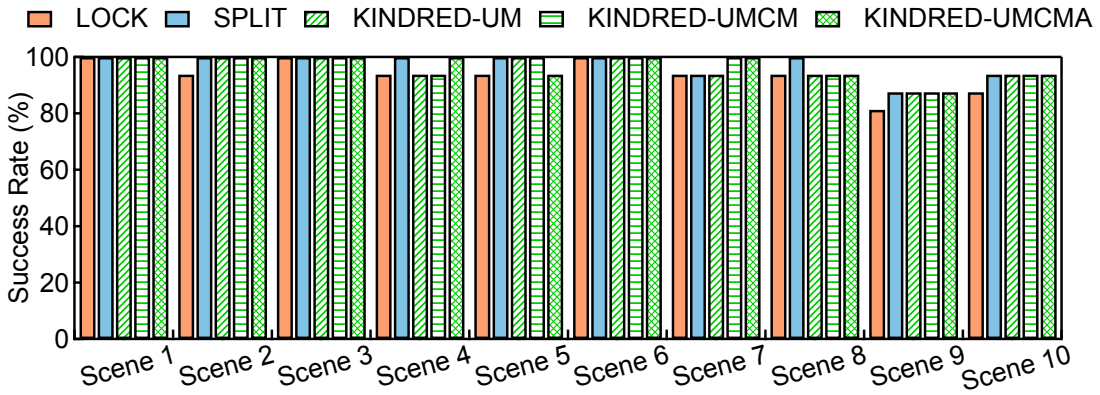
Fig. 6.11: Task pipeline latency comparison.

using Synopsys synthesis and Cadence layout tools with the Silvaco Open-Cell 15nm technology (15n). We incorporate the latency of error detection and correction of both dual-core setup and triple-core setup (Iturbe et al., 2019) into Autoware. All the nodes in Autoware will have extra error detection latency. One node will also have extra error correction latency.

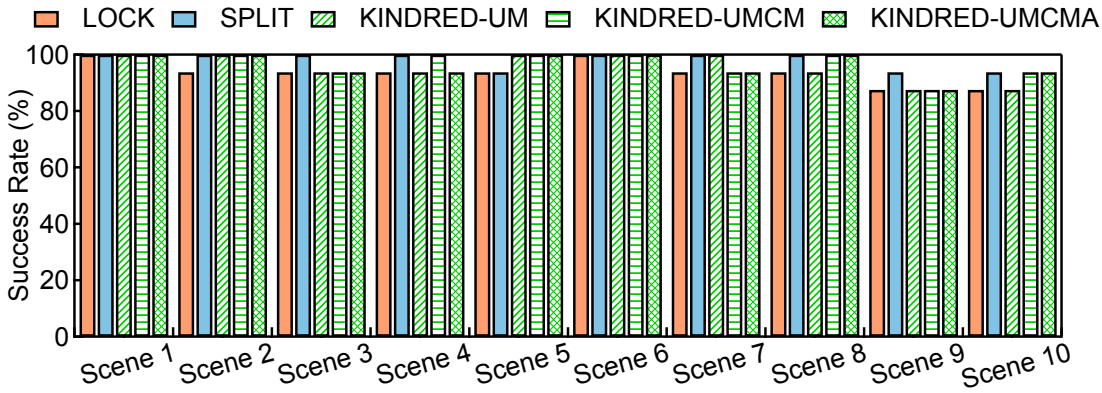
We run the experiments on a machine with 12th Generation Intel Core i9-12900 CPU and NVIDIA RTX 3060 GPU. We bind the software only on the CPU performance cores.

6.7.2 Software Setup

We run a real autonomous machine software stack Autoware and measure its performance. We rely on CARLA to provide simulation of the environments. We monitor the latency of 12 nodes in Autoware and the end-to-end latency of the localization and perception pipelines. The twelve nodes contain two nodes in the sensor preprocessing module, five nodes in the perception module, one node in the localization module and



(a) Mission success rate comparison between baselines and KINDRED with a dual-core setting.



(b) Mission success rate comparison between baselines and KINDRED with a triple-core setting.

Fig. 6.12: Mission success rate comparison.

four nodes in the planning and control module.

Error injection. We inject errors to mimic the SDC events inside the system. Previous work (Gan et al., 2022; Hsiao et al., 2021) show that soft errors, while fundamentally a hardware-level phenomenon, reflect at the software level as corruptions in the communication messages between different nodes. Therefore, one can emulate the effects of soft errors but injecting faults to the inter-node messages, i.e., ROS topics in Autoware. We follow the same methodology here. To ensure that our fault injections are representative, we reuse the same fault values found by Bruam (Gan et al., 2022), which obtained fault ROI topics values under soft errors through hardware-level fault

injections.

Evaluation metric. We use three metrics to evaluate KINDRED. We first present the latency comparison of individual nodes and end-to-end module pipelines. We also create ten different scenarios and run the vehicle inside each scenario of each different setting for sixteen times and report the success rate. To verify the reliability, we use the metric Error Propagation Rate (EPR), which is similar to previous works (Gan et al., 2022). The EPR represents the probability of errors propagate to the output of the autonomous software stack when SDC happen at each individual node. Lower EPR represents higher reliability of the system.

6.7.3 Evaluation Plan

In a typical autonomous machine software stack, a node's fault tolerance can be classified into four different levels: unconditional masking (UM), conditional masking (CM), attenuation (A) and no masking (NM). Our evaluation plan is based on different scheduling strategy of different nodes. We resort to the fault tolerance classification proposed by previous works (Gan et al., 2022).

- LOCK: This baseline represents only lock domain is provided and all the nodes need to run on it.
- SPLIT: This baseline represents only split domain is provided and all the nodes need to run on it. This is the common setting where most autonomous machines are using where no cores are locked.
- KINDRED-UM: In this setting, nodes except UM classification will be running in lock domain, other nodes can run anywhere.
- KINDRED-UMCM: In this setting, nodes with UM and CM classification will can run anywhere, other nodes will be running in lock domain.

- KINDRED-UMCMA: In this setting, only nodes with NM classification will be running in lock domain and all other nodes can run anywhere.

6.8 Evaluation

We first show that KINDRED system brings negligible area overhead and memory overhead (Chapter 6.8.1). We then compare the latency of individual nodes and task pipelines in a typical autonomous machine software and show success rate (Chapter 6.8.2). We evaluate the reliability at last (Chapter 6.8.3)

6.8.1 Overhead Analysis

Chip area overhead. We implement the extra hardware brought by KINDRED. For triple-core lock-step system, where three extra components are included. The majority voter, checker and resynchronization logic in total takes an area of 2167 um^2. Compared to a typical CPU used in an autonomous machine system, ARM A77, the extra logic adds 0.85% area overhead. The area overhead of a dual-core lock-step system is less than 0.2%, as it only needs an extra comparator.

Memory overhead. KINDRED brings extra memory overhead as all the nodes need to save register values and communicate with the kernel module. For Autoware, KINDRED brings 11.5 KB extra memory overhead.

6.8.2 Latency and Success Rate

Latency. We compare the latency of representative nodes of dual-core setting in Fig. 6.10a and triple-core setting in Fig. 6.10b. We plot the y-axis on a logarithmic scale with a base of 10 ($log10$) as the latency difference between different nodes are high. We also show the variance using error bars.

SPLIT has the lowest average latency and LOCK has the highest one. On every node, running in LOCK results in significantly higher latency. The average latency overhead brought by LOCK is 31.5% in the dual-core setting and 26.9% in the triple-core setting. The reason is applying lock-step in the system reduces the available computational resources as well as introduces extra overhead on error detection and correction.

The same conclusions exist in the task pipeline latency. Fig. 6.11a shows the latency of the localization pipeline and perception pipeline. The average latency of localization increases 10.6% and the average latency of perception increase 18.7% in LOCK.

KINDRED significantly reduces the introduced latency overhead. For the dual-core setting, compared to LOCK, KINDRED-UM improves the average latency in individual nodes by 13.5%. Compared to SPLIT, KINDRED-UM only introduces 6.8% average overhead on different nodes. The overhead of KINDRED-UM compared to SPLIT is 0.6% on the localization pipeline and 8.1% on the perception pipeline. The reason is that KINDRED provides two domains in the system. By scheduling nodes with UM classification on the split domain, KINDRED significantly reduces the resource contention and error detection and correction overhead.

KINDRED-UMCM and KINDRED-UMCMA further reduces the overhead as KINDRED schedules more nodes to split domain. Compared to LOCK, KINDRED-UMCMA improves the average node latency by 18.5% and average task pipeline latency by 11.0%. The average task pipeline overhead KINDRED-UMCMA introduces compared to LOCK is only 2.8%.

Success rate. The latency comparison transfers to mission success rate. We show the success rate comparison in Fig. 6.12. SPLIT has the highest success rate. It achieves 100% success rate in most scenarios. The average success rate of SPLIT is 97.5% in the dual-core setting and 98.1% in the triple-core setting. LOCK fails more often. LOCK has an average 93.8% success rate in the dual-core setting and 93.8% success rate in the triple-core setting.

Table 6.1: Error Propagation Rate (EPR) of KINDRED.

MODE	LOCK	SPLIT	UM	UMCM	UMCMA
EPR	0%	16.2%	0%	4.5%	8.3%

KINDRED improves the mission success rate. KINDRED-UM has a mission success rate of 96.3% in dual-core setting and 95.6% in triple-core setting. KINDRED-UMCM has a mission success rate of 96.9% in dual-core setting and 95.6% in triple-core setting. KINDRED-UMCMA has a similar success rate of 96.9% and 96.3%.

6.8.3 Error Propagation Rate

We compare the robustness of the five modes by comparing their EPR. Higher EPR means more errors propagate to the output of the AV software, indicates lower robustness. We show the results in Tbl. 6.1.

Although with the highest latency and lowest success rate, LOCK has the highest robustness and achieve 0% EPR. This is because in LOCK mode, all the nodes run in lock domain and all errors can be detected and corrected. SPLIT has the lowest robustness with highest EPR as all the nodes run in split domain and thus the system has no ability to detect errors. 16.2% of the errors will propagate to the output of the AV software. For KINDRED, KINDRED-UM has the same EPR compared to LOCK. This is because KINDRED-UM schedules only nodes with unconditional masking classification on split domain which are naturally fault tolerant. KINDRED-UMCM and KINDRED-UMCMA both has higher EPR compared to KINDRED-UM. The reason is that scheduling nodes with fault tolerance classification as conditional masking and attenuation on split domain for higher performance is taking the risk that some errors are not detected and recovered as they do not run on lock domain.

6.9 Related Work

Multi-core lock-step CPU design has been employed in specific aerospace applications (Kasap et al., 2021; LaFrieda et al., 2007). Our work aims to enhance system efficiency by implementing a hybrid domain CPU design and scheduling autonomous machine software on it. Rather than using commonly adopted hardware-assisted checkpoint and recovery techniques to improve the efficiency of lock-step systems (Mendon et al., 2012; Doudalis and Prvulovic, 2011; Sorin et al., 2000), we leverage task-level idempotency in autonomous machine software and apply hardware-software collaborative error correction, thereby avoiding the introduction of additional hardware.

Several works, such as Hsiao et al. (Hsiao et al., 2021), Gan et al. (Gan et al., 2022), Koopman (Koopman and Wagner, 2017), and Basargan et al. (Basargan et al., 2021), have proposed identifying divergence in fault tolerance classification between different nodes in autonomous machine software. This work is motivated by the same insights, and we aim to harness the inherent fault tolerance of the software to schedule nodes with high reliability in the split domain. By doing so, we can address resource contention and enhance performance, ultimately improving the efficiency and reliability of the autonomous machine software.

This work focuses on lock-step CPU design. We notice that currently multiple different accelerators such as GPU, DSP and NPU have been utilized in building hardware for autonomous machines (Yu et al., 2020; Krishnan et al., 2022; Suleiman et al., 2019, 2018; Talpes et al., 2020; Yang et al., 2021). How to enable lock-step accelerator design is also a challenging topic and has not been well studied yet.

6.10 Discussion and Future work

Lock-step CPU design methodology. In this work, we apply the comparator on the output signals of the CPU logic cores. In our implementation, we do not need to

add extra pipeline states and our solution can be integrated into existing hardware easily. The downside of our design is that the architectural states and micro-architectural states may have been polluted as we do not compare the architectural states table after each instruction. We mitigate this effect by constantly saving correct register values to the memory and restore from them to ensure error correction. A comparison between checking the output signals of CPU cores and checking the internal architectural states table could help us more in understanding different implementation methodologies of lock-step CPU designs.

Static and dynamic multi-domain design. We propose a static multi-domain design in this work. That is, the domain separation of lock and split is static and can not be changed during runtime. This form of design does not allow runtime reconfiguration. However, the workload of autonomous machines varies significantly as the scenario changes. The distribution of workload scheduled on lock-domain and split-domain may change when the vehicle gets out of the highway and start parking inside the city. A dynamic multi-domain lock-step CPU design that allows reconfiguration is an interesting line of future work. How to achieve low latency reconfiguration between lock cores and split cores and the condition of reconfiguration are the key challenges.

7 Retrospective and Future Work

This chapter provides retrospective and the potential future directions of my dissertation work. I first introduces two principles developed from my work in Chapter 7.1. I further propose future directions of my dissertation in Chapter 7.2.

7.1 Retrospective

My thesis represents the initial step in developing a reliable computing system for autonomous machines. The objective of my research is to address the reliability concerns within both the perception module and the overall software of autonomous machines. In summary, this thesis elucidates two primary principles:

- Performance and Reliability are Equally Important: For autonomous machines, performance and reliability are two equally significant metrics. Both measurements play critical roles in ensuring the safety and usability of these machines. A majority of the current research primarily concentrates on enhancing performance while disregarding reliability or vice versa. In this thesis, I establish that such approaches are inadequate. I illustrate, through empirical evidence, that implementing a suboptimal method to guarantee reliability can negatively impact

performance and subsequently affect the success rate of missions undertaken by autonomous machines.

- Exploring and Utilizing Algorithmic Characteristics: Conventional techniques for ensuring fault tolerance are predominantly general and applicable to various types of algorithms. In my dissertation, a principal insight emphasizes the need for system designers to investigate and capitalize on algorithmic characteristics to provide customized protection for the computing stack. One of the key tenets underlying my work is the close integration of protection with application-specific characteristics. For instance, the BRAUM method initially analyzes the software stack and generates a distinct inherent fault tolerance classification, allowing KINDRED to subsequently propose a multi-domain architecture addressing both reliability and performance.

7.2 Future Work

The future of computing systems for autonomous machines is undoubtedly exhilarating. I posit that we are entering an era in which autonomous machines assume significantly more prominent roles within human society. Informed by the principles outlined in the preceding section, I propose several critical future directions for the computational aspects of autonomous machines that I believe warrant further exploration.

- Customized Split-Lock Architecture for Accelerators: Owing to the swift evolution of algorithms employed in autonomous machines, various accelerators, such as graphic processing units (GPUs) and neuron processing units (NPUs), have become integral components of these machines' computing systems. Presently, most fault tolerance techniques applied to accelerators involve allocating additional silicon resources to introduce redundancy. Inspired by the KINDRED ap-

proach, I propose the development of a customized split-lock architecture specifically tailored for accelerators.

For instance, considering the distinct computational flow of deep neural networks, a lock-step NPU need not conduct a cycle-by-cycle comparison of the partial sum. A delayed comparison when results are committed to the off-chip DRAM is feasible. Concurrently, various deep learning algorithms exhibit different levels of inherent fault tolerance. Developing a heterogeneous split-lock hardware design and a corresponding scheduler for NPUs presents an intriguing topic for further exploration.

- Autonomous Machine Software 2.0: The landscape of autonomous machine software is constantly evolving. In my dissertation, I primarily examine traditional autonomous machine software, characterized by well-defined modules. However, as large generative models experience significant advancements (Shen et al., 2023; Zhang and Li, 2021), a trend has emerged wherein a single model supplants the traditional complex computational graph (Liu et al., 2022; Li et al., 2022a; Liang et al., 2023; Li et al., 2022c).

  This emerging trend in autonomous machine software is revolutionizing the field. Firstly, numerous algorithms within planning and control modules are evolving from state machine-driven approaches to those driven by vast data and computation. Consequently, a new hardware architecture is required to accommodate this shift in software design. Simultaneously, the landscape of reliability concerns is also changing. Traditionally, planning and control modules were more susceptible to soft errors and software bugs. However, the transformed software architecture may exhibit increased robustness against soft errors while becoming more vulnerable to adversarial attacks.

8 Conclusion

In this dissertation, I endeavor to address the reliability concerns of computing systems within autonomous machines. My objective is to transcend the conventional performance-reliability trade-off, safeguarding the system while minimizing overhead. In pursuit of this goal, I strive to tackle two primary challenges. The first challenge involves constructing a dependable perception module capable of withstanding adversarial attacks. The second challenge seeks to establish a reliable computing system specifically tailored for autonomous machines.

The first project encompasses an efficient adversarial example detector and a dynamic network topology capable of processing both standard and adversarial images. The adversarial example detector, PTOLEMY, attains high detection accuracy (0.9 AUC) with minimal overhead (an additional 2.1% latency). In conjunction with PTOLEMY, the dynamic network MORPHADNET achieves 0.92 accuracy for standard images and 0.89 accuracy for adversarial images on the CIFAR-10 dataset.

The second project consists of an inherent fault tolerance classification of autonomous machine software stack, BRAUM, and a heterogeneous split-lock CPU architecture design KINDRED that utilizes the classification of BRAUM. KINDRED is able to correct all the errors it encounters while only introduce 13.5% latency overhead.

Bibliography

15NM OPEN-CELL LIBRARY. http://www.si2.org/open-cell-library/. URL http://www.si2.org/open-cell-library/.

Baidu Apollo team (2017), Apollo: Open Source Autonomous Driving, howpublished = https://github.com/apolloauto/apollo, note = Accessed: 2019-02-11.

NVIDIA Reveals Xavier SOC Details. https://bit.ly/2qq0TWp. URL https://www.forbes.com/sites/moorinsights/2018/08/24/nvidia-reveals-xavier-soc-details/amp/.

Ahmed Fathy Mohammed Abdelrazek. Exception and interrupt handling in arm. *Universität Stuttgart, Report*, 2006.

Jacob A Abraham and Daniel P Siewiorek. An algorithm for the accurate reliability evaluation of triple modular redundancy networks. *IEEE Transactions on Computers*, 100(7):682–692, 1974.

Evan Ackerman. Lidar that will make self-driving cars affordable [news]. *IEEE Spectrum*, 53(10):14–14, 2016.

Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P Jouppi, and James E Smith. Configurable isolation: building high availability systems with commodity multi-core processors. *ACM SIGARCH Computer Architecture News*, 35(2):470–481, 2007.

Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60: 19–31, 2016.

Naveed Akhtar and Ajmal Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *Ieee Access*, 6:14410–14430, 2018.

Ahmed Aldahdooh, Wassim Hamidouche, and Olivier Déforges. Selective and features based adversarial example detection. *arXiv preprint arXiv:2103.05354*, 2021.

Vicki H Allan, Reese B Jones, Randall M Lee, and Stephen J Allan. Software pipelining. *ACM Computing Surveys (CSUR)*, 27(3):367–432, 1995.

Maksym Andriushchenko, Francesco Croce, Nicolas Flammarion, and Matthias Hein. Square attack: a query-efficient black-box adversarial attack via random search. In *European Conference on Computer Vision*, pages 484–501. Springer, 2020.

Lorena Anghel, Dan Alexandrescu, and Michael Nicolaidis. Evaluation of a soft error tolerance technique based on time and/or space redundancy. In *Proceedings 13th Symposium on Integrated Circuits and Systems Design (Cat. No. PR00843)*, pages 237–242. IEEE, 2000.

Fatemeh Ayatolahi, Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson. A study of the impact of single bit-flip and double bit-flip errors on program execution. In *Computer Safety, Reliability, and Security: 32nd International Conference, SAFECOMP 2013, Toulouse, France, September 24-27, 2013. Proceedings 32*, pages 265–276. Springer, 2013.

Thomas Ball and James R Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

Vitor Bandeira, Isadora Oliveira, Felipe da Rosa, Ricardo Reis, and Luciano Ost. Soft error reliability analysis of autonomous vehicles software stack. In *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 253–254. IEEE, 2019.

Hakan Basargan, András Mihály, and Péter Gáspár. Fault-tolerant trajectory tracking control for autonomous vehicle based on camera and gps. In *2021 European Control Conference (ECC)*, pages 473–478. IEEE, 2021.

Robert Baumann. Soft errors in advanced computer systems. *IEEE design & test of computers*, 22(3):258–266, 2005a.

Robert C Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability*, 5(3):305–316, 2005b.

Mark P Baze, Steven P Buchner, and Dale McMorrow. A digital cmos design technique for seu hardening. *IEEE Transactions on Nuclear Science*, 47(6):2603–2608, 2000.

Irwan Bello, Barret Zoph, Ashish Vaswani, Jonathon Shlens, and Quoc V Le. Attention augmented convolutional networks. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 3286–3295, 2019.

Philipp Benz, Chaoning Zhang, and In So Kweon. Batch normalization increases adversarial vulnerability and decreases adversarial transferability: A non-robust feature perspective. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7818–7827, 2021.

Maarten M Bezemer and Jan F Broenink. Connecting ros to a real-time control framework for embedded computing. In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–6. IEEE, 2015.

B Bhuva. Soft error trends in advanced silicon technology nodes. In *2018 IEEE International Electron Devices Meeting (IEDM)*, pages 34–4. IEEE, 2018.

Michael Bloesch, Sammy Omari, Marco Hutter, and Roland Siegwart. Robust visual inertial odometry using a direct ekf-based approach. In *2015 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 298–304. IEEE, 2015.

Jason Blome, Scott Mahlke, Daryl Bradley, and Krisztián Flautner. A microarchitectural analysis of soft error propagation in a production-level embedded microprocessor. In *In Proceedings of the First Workshop on Architecture Reliability*. Citeseer, 2005.

Demid Borodin and Ben HH Juurlink. Protective redundancy overhead reduction using instruction vulnerability factor. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 319–326, 2010.

Tamal Bose and Francois Meyer. *Digital signal and image processing*. John Wiley & Sons, Inc., 2003.

Neal E Boudette. 'it happened so fast': Inside a fatal tesla autopilot crash, 2021a. URL https://www.irishtimes.com/life-and-style/motors/it-happened-so-fast-inside-a-fatal-tesla-autopilot-crash-1.4650265.

Neal E Boudette. U.s. will investigate tesla's autopilot system over crashes with emergency vehicles, 2021b. URL https://www.nytimes.com/2021/08/16/business/tesla-autopilot-nhtsa.html.

John Bradshaw, Alexander G de G Matthews, and Zoubin Ghahramani. Adversarial examples, uncertainty, and transfer testing robustness in gaussian process hybrid deep networks. *arXiv preprint arXiv:1707.02476*, 2017.

Jacob Buckman, Aurko Roy, Colin Raffel, and Ian Goodfellow. Thermometer encoding: One hot way to resist adversarial examples. 2018.

Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 3–14. ACM, 2017a.

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, 2017b.

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, 2017c.

Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, Aleksander Madry, and Alexey Kurakin. On evaluating adversarial robustness. *arXiv preprint arXiv:1902.06705*, 2019.

Anirban Chakraborty, Manaar Alam, Vishal Dey, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. Adversarial attacks and defences: A survey. *arXiv preprint arXiv:1810.00069*, 2018.

Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58, 2009.

Pohua P Chang and WW Hwu. Trace selection for compiling large c application programs to microcode. In *Proceedings of the 21st annual workshop on Microprogramming and microarchitecture*, pages 21–29. IEEE Computer Society Press, 1988.

Dr. Indranil Chatterjee. From mosfets to finfets - the soft error scaling trends. https://radnext.web.cern.ch/blog/from-mosfets-to-finfets/.

Ren Chen, Sruja Siriyal, and Viktor Prasanna. Energy and memory efficient mapping of bitonic sorting on fpga. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 240–249. ACM, 2015.

Shuyu Cheng, Yinpeng Dong, Tianyu Pang, Hang Su, and Jun Zhu. Improving black-box adversarial attacks with a transfer-based prior. *arXiv preprint arXiv:1906.06919*, 2019.

David R Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):215–232, 1958.

Patricia Craja, Alisa Kim, and Stefan Lessmann. Deep learning for detecting financial statement fraud. *Decision Support Systems*, 139:113421, 2020.

Francesco Croce and Matthias Hein. Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In *International conference on machine learning*, pages 2206–2216. PMLR, 2020.

Marc De Kruijf and Karthikeyan Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12. IEEE, 2013.

Ádria Barros de Oliveira, Gennaro Severino Rodrigues, and Fernanda Lima Kastensmidt. Analyzing lockstep dual-core arm cortex-a9 soft error mitigation in freertos applications. In *Proceedings of the 30th Symposium on Integrated Circuits and Systems Design: Chip on the Sands*, pages 84–89, 2017.

Adria Barros de Oliveira, Gennaro Severino Rodrigues, Fernanda Lima Kastensmidt, Nemitala Added, Eduardo LA Macchione, Vitor AP Aguiar, Nilberto H Medina, and Marcilei AG Silveira. Lockstep dual-core arm a9: Implementation and resilience analysis under heavy ion-induced soft errors. *IEEE Transactions on Nuclear Science*, 65(8):1783–1790, 2018.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

Guneet S Dhillon, Kamyar Azizzadenesheli, Zachary C Lipton, Jeremy Bernstein, Jean Kossaifi, Aran Khanna, and Anima Anandkumar. Stochastic activation pruning for robust adversarial defense. *arXiv preprint arXiv:1803.01442*, 2018.

Renwei Dian, Shutao Li, Anjing Guo, and Leyuan Fang. Deep hyperspectral image sharpening. *IEEE transactions on neural networks and learning systems*, 29(11): 5345–5355, 2018.

Michael Ditty. Nvidia orin system-on-chip. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–17. IEEE Computer Society, 2022.

Anand Dixit and Alan Wood. The impact of new technology on soft error rates. In *2011 International Reliability Physics Symposium*, pages 5B–4. IEEE, 2011.

Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245*, 2021.

Hadi M Dolatabadi, Sarah Erfani, and Christopher Leckie. Advflow: Inconspicuous black-box adversarial attacks using normalizing flows. *arXiv preprint arXiv:2007.07435*, 2020.

Robert John Donovan, Robert Ralph Roediger, and William Jon Schmidt. Profile driven optimization of frequently executed paths with inlining of code fragment (one or more lines of code from a child procedure to a parent procedure), June 6 2000. US Patent 6,072,951.

Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. In *Conference on robot learning*, pages 1–16. PMLR, 2017.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Ioannis Doudalis and Milos Prvulovic. Hare++: Hardware assisted reverse execution revisited. *Georgia Institute of Technology*, pages 1–8, 2011.

Xinxin Du, Marcelo H Ang, and Daniela Rus. Car detection for autonomous vehicle: Lidar and vision fusion approach through deep learning framework. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 749–754. IEEE, 2017.

Mihai Dusmanu, Johannes L Schönberger, Sudipta N Sinha, and Marc Pollefeys. Privacy-preserving image features via adversarial affine subspace embeddings. *arXiv preprint arXiv:2006.06634*, 2020.

James Elliott, Frank Mueller, Frank Stoyanov, and Clayton Webster. Quantifying the impact of single bit flips on floating point arithmetic. Technical report, North Carolina State University. Dept. of Computer Science, 2013.

Christian Engelmann, Hong H Ong, and Stephen L Scott. The case for modular redundancy in large-scale high performance computing systems. In *Proceedings of the 8th IASTED international conference on parallel and distributed computing and networks (PDCN)*, pages 189–194, 2009.

Marsel Faizullin, Anastasiia Kornilova, and Gonzalo Ferrer. Open-source lidar time synchronization system by mimicking gnss-clock. In *2022 IEEE International Sym-*

posium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS), pages 1–5. IEEE, 2022.

Haoyang Fan, Fan Zhu, Changchun Liu, Liangliang Zhang, Li Zhuang, Dong Li, Weicheng Zhu, Jiangtao Hu, Hongye Li, and Qi Kong. Baidu apollo em motion planner. *arXiv preprint arXiv:1807.08048*, 2018.

Bo Fang, Qining Lu, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. epvf: An enhanced program vulnerability factor methodology for cross-layer resilience analysis. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 168–179. IEEE, 2016.

Weiwei Fang, Xin Li, Ping Zhou, Jingwen Yan, Dazhi Jiang, and Teng Zhou. Deep learning anti-fraud model for internet loan: Where we are going. *IEEE Access*, 9: 9777–9784, 2021.

Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE transactions on computers*, (7):478–490, 1981.

Scott Freitas, Andrew Wicker, Duen Horng Chau, and Joshua Neil. D2m: Dynamic defense and modeling of adversarial movement in networks. In *Proceedings of the 2020 SIAM International Conference on Data Mining*, pages 541–549. SIAM, 2020.

Yiming Gan, Yuxian Qiu, Jingwen Leng, Minyi Guo, and Yuhao Zhu. Ptolemy: Architecture support for robust deep learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 241–255. IEEE, 2020.

Yiming Gan, Yu Bo, Boyuan Tian, Leimeng Xu, Wei Hu, Shaoshan Liu, Qiang Liu, Yanjun Zhang, Jie Tang, and Yuhao Zhu. Eudoxus: Characterizing and accelerating localization in autonomous machines industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 827–840. IEEE, 2021.

Yiming Gan, Paul Whatmough, Jingwen Leng, Bo Yu, Shaoshan Liu, and Yuhao Zhu. Braum: Analyzing and protecting autonomous machine software stack. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 85–96. IEEE, 2022.

Hongbo Gao, Bo Cheng, Jianqiang Wang, Keqiang Li, Jianhui Zhao, and Deyi Li. Object classification using cnn-based fusion of vision and lidar in autonomous vehicle environment. *IEEE Transactions on Industrial Informatics*, 14(9):4224–4231, 2018a.

Ji Gao, Jack Lanchantin, Mary Lou Soffa, and Yanjun Qi. Black-box generation of adversarial text sequences to evade deep learning classifiers. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 50–56. IEEE, 2018b.

Evrard Garcelon, Baptiste Roziere, Laurent Meunier, Jean Tarbouriech, Olivier Teytaud, Alessandro Lazaric, and Matteo Pirotta. Adversarial attacks on linear contextual bandits. *arXiv preprint arXiv:2002.03839*, 2020.

Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, Chen, and Qi Alfred. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, pages 385–396, 2020.

Shafi Goldwasser, Michael P Kim, Vinod Vaikuntanathan, and Or Zamir. Planting undetectable backdoors in machine learning models. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 931–942. IEEE, 2022.

Zhitao Gong, Wenlu Wang, and Wei-Shinn Ku. Adversarial and clean data are not twins. *arXiv preprint arXiv:1704.04960*, 2017.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014a.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014b.

Shixiang Gu and Luca Rigazio. Towards deep neural network architectures robust to adversarial examples. *arXiv preprint arXiv:1412.5068*, 2014.

Chuan Guo, Mayank Rana, Moustapha Cisse, and Laurens Van Der Maaten. Countering adversarial images using input transformations. *arXiv preprint arXiv:1711.00117*, 2017.

Chuan Guo, Jacob Gardner, Yurong You, Andrew Gordon Wilson, and Kilian Weinberger. Simple black-box adversarial attacks. In *International Conference on Machine Learning*, pages 2484–2493. PMLR, 2019.

Ying Guo, Xingxing Wei, Guoqiu Wang, and Bo Zhang. Meaningful adversarial stickers for face recognition in physical world. *arXiv preprint arXiv:2104.06728*, 2021.

Abhishek Gupta, Alagan Anpalagan, Ling Guan, and Ahmed Shaharyar Khwaja. Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues. *Array*, 10:100057, 2021.

Gor Hakobyan and Bin Yang. High-performance automotive radar: A review of signal processing algorithms and modulation schemes. *IEEE Signal Processing Magazine*, 36(5):32–44, 2019.

Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, et al. A survey on vision transformer. *IEEE transactions on pattern analysis and machine intelligence*, 45(1):87–110, 2022.

Sudheendra Hangal and Monica S Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301, 2002.

Derek Hatley and Imtiaz Pirbhai. *Strategies for real-time system specification*. Addison-Wesley, 2013.

Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*, pages 207–218. IEEE, 2016.

Warren He, James Wei, Xinyun Chen, Nicholas Carlini, and Dawn Song. Adversarial example defense: Ensembles of weak defenses are not strong. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

Zhezhi He, Adnan Siraj Rakin, and Deliang Fan. Parametric noise injection: Trainable randomness to improve deep neural network robustness against adversarial attack. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 588–597, 2019.

Jeff Hecht. Lidar for self-driving cars. *Optics and Photonics News*, 29(1):26–33, 2018.

Keigo Hirakawa and Thomas W Parks. Joint demosaicing and denoising. *IEEE Transactions on Image Processing*, 15(8):2146–2157, 2006.

Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. Deep models under the gan: information leakage from collaborative deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 603–618, 2017.

Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 9–16, 2021.

Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174. ACM, 2008.

Yu-Shun Hsiao, Zishen Wan, Tianyu Jia, Radhika Ghosal, Arijit Raychowdhury, David Brooks, Gu-Yeon Wei, and Vijay Janapa Reddi. Mavfi: An end-to-end fault analysis framework with anomaly detection and recovery for micro aerial vehicles. *arXiv preprint arXiv:2105.12882*, 2021.

Ting-Kuei Hu, Tianlong Chen, Haotao Wang, and Zhangyang Wang. Triple wins: Boosting accuracy, robustness and efficiency together by enabling input-adaptive inference. *arXiv preprint arXiv:2002.10025*, 2020a.

Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. Deepsniffer: A dnn model extraction framework based on learning architectural hints. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 385–399, 2020b.

Jin Huang and Charles X Ling. Using auc and accuracy in evaluating learning algorithms. *IEEE Transactions on knowledge and Data Engineering*, 17(3):299–310, 2005.

Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies. *arXiv preprint arXiv:1702.02284*, 2017.

Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.

Xabier Iturbe, Balaji Venu, and Emre Ozer. Soft error vulnerability assessment of the real-time safety-related arm cortex-r5 cpu. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 91–96. IEEE, 2016a.

Xabier Iturbe, Balaji Venu, Emre Ozer, and Shidhartha Das. A triple core lock-step (tcls) arm® cortex®-r5 processor for safety-critical and ultra-reliable applications.

In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 246–249. IEEE, 2016b.

Xabier Iturbe, Balaji Venu, Emre Ozer, Jean-Luc Poupat, Gregoire Gimenez, and Hans-Ulrich Zurek. The arm triple core lock-step (tcls) processor. *ACM Transactions on Computer Systems (TOCS)*, 36(3):1–30, 2019.

Hrag-Harout Jebamikyous and Rasha Kashef. Autonomous vehicles perception (avp) using deep learning: Modeling, assessment, and challenges. *IEEE Access*, 10:10523–10535, 2022.

Xu Jia, Bert De Brabandere, Tinne Tuytelaars, and Luc V Gool. Dynamic filter networks. *Advances in neural information processing systems*, 29:667–675, 2016.

Mathai Joseph and Paritosh Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

Nidhi Kalra and Susan M Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice*, 94:182–193, 2016.

Harini Kannan, Alexey Kurakin, and Ian Goodfellow. Adversarial logit pairing. *arXiv preprint arXiv:1803.06373*, 2018.

Server Kasap, Eduardo Weber Wächter, Xiaojun Zhai, Shoaib Ehsan, and Klaus D McDonald-Maier. Novel lockstep-based fault mitigation approach for socs with roll-back and roll-forward recovery. *Microelectronics Reliability*, 124:114297, 2021.

F Lima Kastensmidt, Luca Sterpone, Luigi Carro, and M Sonza Reorda. On the optimal design of triple modular redundancy logic for sram-based fpgas. In *Design, Automation and Test in Europe*, pages 1290–1295. IEEE, 2005.

Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296, 2018.

Jackie Kay and Adolfo Rodriguez Tsouroukdissian. Real-time control in ros and ros 2.0. *ROSCon15*, 2015.

Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. Transformers in vision: A survey. *ACM computing surveys (CSUR)*, 54(10s):1–41, 2022.

Byung Kook Kim. Reliability analysis of real-time controllers with dual-modular temporal redundancy. In *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No. PR00306)*, pages 364–371. IEEE, 1999.

Eric P Kim and Naresh R Shanbhag. Soft n-modular redundancy. *IEEE Transactions on Computers*, 61(3):323–336, 2010.

Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Intrusion recovery using selective re-execution. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

Donald E Knuth. *Art of computer programming, volume 3: Sorting and Searching*. Addison-Wesley Professional, 2014.

Dirk Koch and Jim Torresen. Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 45–54. ACM, 2011.

Stepan Komkov and Aleksandr Petiushko. Advhat: Real-world adversarial attack on arcface face id system. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 819–826. IEEE, 2021.

Philip Koopman and Michael Wagner. Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety*, 4(1):15–24, 2016.

Philip Koopman and Michael Wagner. Autonomous vehicle safety: An interdisciplinary challenge. *IEEE Intelligent Transportation Systems Magazine*, 9(1):90–96, 2017.

David Kortenkamp, R Peter Bonasso, and Robin Murphy. *Artificial intelligence and mobile robots: case studies of successful robot systems*. MIT Press, 1998.

Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Sabrina Neuman, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. Automatic domain-specific soc design for autonomous unmanned aerial vehicles. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 300–317. IEEE, 2022.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-isa heterogeneous multi-core architectures: The potential for

processor power reduction. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 81–92. IEEE, 2003.

Rakesh Kumar, Dean M Tullsen, Parthasarathy Ranganathan, Norman P Jouppi, and Keith I Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. *ACM SIGARCH Computer Architecture News*, 32(2):64, 2004.

Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

Christopher LaFrieda, Engin Ipek, Jose F Martinez, and Rajit Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 317–326. IEEE, 2007.

Leon Lantz. Soft errors induced by alpha particles. *IEEE Transactions on Reliability*, 45(2):174–179, 1996.

Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

Gim Hee Lee, Friedrich Fraundorfer, and Marc Pollefeys. Structureless pose-graph loop-closure with a multi-camera system on a self-driving car. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 564–571. IEEE, 2013.

J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, and V. J. Reddi. Safe limits on voltage reduction efficiency in gpus: A direct measurement approach. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 294–307, 2015.

J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, Q. Chen, M. Guo, and V. Janapa Reddi. Asymmetric resilience: Exploiting task-level idempotency for transient error recovery in accelerator-based systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 44–57, 2020.

Henrietta Lengyel, Viktor Remeli, and Zsolt Szalay. A collection of easily deployable adversarial traffic sign stickers. *at-Automatisierungstechnik*, 69(6):511–523, 2021.

Alexander Levine and Soheil Feizi. (de) randomized smoothing for certifiable defense against patch attacks. *arXiv preprint arXiv:2002.10733*, 2020.

Hongyang Li, Chonghao Sima, Jifeng Dai, Wenhai Wang, Lewei Lu, Huijie Wang, Enze Xie, Zhiqi Li, Hanming Deng, Hao Tian, et al. Delving into the devils of bird's-eye-view perception: A review, evaluation and recipe. *arXiv preprint arXiv:2209.05324*, 2022a.

Juncheng Li, Frank Schmidt, and Zico Kolter. Adversarial camera stickers: A physical camera-based attack on deep learning systems. In *International Conference on Machine Learning*, pages 3896–3904. PMLR, 2019.

Peizhao Li, Pu Wang, Karl Berntorp, and Hongfu Liu. Exploiting temporal relations on radar perception for autonomous driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 17071–17080, 2022b.

Qizhang Li, Yiwen Guo, and Hao Chen. Practical no-box adversarial attacks against dnns. *arXiv preprint arXiv:2012.02525*, 2020.

Tuo Li, Jude Angelo Ambrose, Roshan Ragel, and Sri Parameswaran. Processor design for soft errors: Challenges and state of the art. *ACM Computing Surveys (CSUR)*, 49 (3):1–44, 2016.

Xin Li, Bahadir Gunturk, and Lei Zhang. Image demosaicing: A systematic survey. In *Visual Communications and Image Processing 2008*, volume 6822, pages 489–503. SPIE, 2008.

Zhiqi Li, Wenhai Wang, Hongyang Li, Enze Xie, Chonghao Sima, Tong Lu, Yu Qiao, and Jifeng Dai. Bevformer: Learning bird's-eye-view representation from multi-camera images via spatiotemporal transformers. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part IX*, pages 1–18. Springer, 2022c.

Zhixuan Liang, Yao Mu, Mingyu Ding, Fei Ni, Masayoshi Tomizuka, and Ping Luo. Adaptdiffuser: Diffusion models as adaptive self-evolving planners. *arXiv preprint arXiv:2302.01877*, 2023.

Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

Sheng Lin, Yong-Bin Kim, and Fabrizio Lombardi. Design and performance evaluation of radiation hardened latches for nanoscale cmos. *IEEE transactions on very large scale integration (VLSI) systems*, 19(7):1315–1319, 2010.

Wei-An Lin, Chun Pong Lau, Alexander Levine, Rama Chellappa, and Soheil Feizi. Dual manifold adversarial robustness: Defense against lp and non-lp adversarial attacks. *arXiv preprint arXiv:2009.02470*, 2020.

Daniel Liu, Ronald Yu, and Hao Su. Extending adversarial attacks and defenses to deep 3d point cloud classifiers. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 2279–2283. IEEE, 2019a.

Liangkai Liu, Sidi Lu, Ren Zhong, Baofu Wu, Yongtao Yao, Qingyang Zhang, and Weisong Shi. Computing systems for autonomous driving: State of the art and challenges. *IEEE Internet of Things Journal*, 8(8):6469–6486, 2020.

Runze Liu, Jianlei Yang, Yiran Chen, and Weisheng Zhao. eslam: An energy-efficient accelerator for real-time orb-slam on fpga platform. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019b.

Weizhuang Liu, Bo Yu, Yiming Gan, Qiang Liu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. Archytas: A framework for synthesizing and dynamically optimizing accelerators for robotic localization. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 479–493, 2021.

Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. *arXiv preprint arXiv:1611.02770*, 2016.

Zhijian Liu, Haotian Tang, Alexander Amini, Xinyu Yang, Huizi Mao, Daniela Rus, and Song Han. Bevfusion: Multi-task multi-sensor fusion with unified bird's-eye view representation. *arXiv preprint arXiv:2205.13542*, 2022.

Atieh Lotfi, Saurabh Hukerikar, Keshav Balasubramanian, Paul Racunas, Nirmal Saxena, Richard Bramley, and Yanxiang Huang. Resiliency of automotive object detection networks on gpu architectures. In *2019 IEEE International Test Conference (ITC)*, pages 1–9. IEEE, 2019.

Jiajun Lu, Theerasit Issaranon, and David Forsyth. Safetynet: Detecting and rejecting adversarial examples robustly. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 446–454, 2017.

Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM journal of research and development*, 6(2):200–209, 1962.

Chunchuan Lyu, Kaizhu Huang, and Hai-Ning Liang. A unified gradient regularization family for adversarial examples. In *2015 IEEE international conference on data mining*, pages 301–309. IEEE, 2015.

Jiaqi Ma, Shuangrui Ding, and Qiaozhu Mei. Towards more practical adversarial attacks on graph neural networks. *arXiv preprint arXiv:2006.05057*, 2020.

Shiqing Ma and Yingqi Liu. Nic: Detecting adversarial samples with neural network invariant checking. In *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS 2019)*, 2019.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

Ashwin A Mendon, Ron Sass, Zachary K Baker, and Justin L Tripp. Design and implementation of a hardware checkpoint/restart core. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, pages 1–6. IEEE, 2012.

Xuelian Meng, Nate Currit, and Kaiguang Zhao. Ground filtering algorithms for airborne lidar data: A review of critical issues. *Remote Sensing*, 2(3):833–860, 2010.

Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. On detecting adversarial perturbations. *arXiv preprint arXiv:1702.04267*, 2017.

David J Miller, Zhen Xiang, and George Kesidis. Adversarial learning targeting deep neural network classification: A comprehensive review of defenses against attacks. *Proceedings of the IEEE*, 108(3):402–433, 2020.

Takeru Miyato, Andrew M Dai, and Ian Goodfellow. Adversarial training methods for semi-supervised text classification. *arXiv preprint arXiv:1605.07725*, 2016.

Antonio Luis Montealegre, María Teresa Lamelas, and Juan De La Riva. A comparison of open-source lidar filtering algorithms in a mediterranean forest environment. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 8(8): 4072–4085, 2015.

Joanna Moody, Nathaniel Bailey, and Jinhua Zhao. Public perceptions of autonomous vehicle safety: An international comparison. *Safety science*, 121:634–650, 2020.

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016a.

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016b.

Mukesh C Motwani, Mukesh C Gadiya, Rakhi C Motwani, and Frederick C Harris. Survey of image denoising techniques. In *Proceedings of GSPX*, volume 27, pages 27–30. Proceedings of GSPX, 2004.

Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting networks on fpgas. *The VLDB Journal—The International Journal on Very Large Data Bases*, 21(1):1–23, 2012.

Shubhendu S Mukherjee, Christopher Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 29–40. IEEE, 2003.

Shubhendu S Mukherjee, Joel Emer, and Steven K Reinhardt. The soft error problem: An architectural perspective. In *11th International Symposium on High-Performance Computer Architecture*, pages 243–247. IEEE, 2005.

Shubu Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2011.

Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE transactions on robotics*, 31(5):1147–1163, 2015.

D Darian Muresan and Thomas W Parks. Adaptive principal components and image denoising. In *Proceedings 2003 International Conference on Image Processing (Cat. No. 03CH37429)*, volume 1, pages I–101. IEEE, 2003.

Junichi Nakamura. *Image sensors and signal processing for digital still cameras*. CRC press, 2017.

Preetum Nakkiran. Adversarial robustness may be at odds with simplicity. *arXiv preprint arXiv:1901.00532*, 2019.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011.

Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 427–436, 2015.

Michael Nicolaidis. *Soft errors in modern electronic systems*, volume 41. Springer Science & Business Media, 2010.

Bogdan Nicolescu, Yvon Savaria, and Raoul Velazco. Software detection mechanisms providing full coverage against single bit-flip faults. *IEEE Transactions on Nuclear science*, 51(6):3510–3518, 2004.

Julia Nitsch, Masha Itkina, Ransalu Senanayake, Juan Nieto, Max Schmidt, Roland Siegwart, Mykel J Kochenderfer, and Cesar Cadena. Out-of-distribution detection for automotive perception. In *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, pages 2938–2943. IEEE, 2021.

Jeffrey Oplinger and Monica S Lam. Enhancing software reliability with speculative threads. *ACM SIGARCH Computer Architecture News*, 30(5):184–196, 2002.

G. Papadimitriou, A. Chatzidimitriou, D. Gizopoulos, V. J. Reddi, J. Leng, B. Salami, O. S. Unsal, and A. C. Kestelman. Exceeding conservative limits: A consolidated analysis on modern hardware margins. *IEEE Transactions on Device and Materials Reliability*, 20(2):341–350, 2020.

Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016a.

Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387. IEEE, 2016b.

Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 582–597. IEEE, 2016c.

Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, et al. Deep face recognition. In *bmvc*, volume 1, page 6, 2015.

Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer networks*, 51(12):3448–3470, 2007.

Mohini Patil, Xuyu Wang, Xiangyu Wang, and Shiwen Mao. Adversarial attacks on deep learning-based floor classification and indoor localization. In *Proceedings of the 3rd ACM Workshop on Wireless Security and Machine Learning*, pages 7–12, 2021.

Ebberth L Paula, Marcelo Ladeira, Rommel N Carvalho, and Thiago Marzagao. Deep learning anomaly detection as support fraud investigation in brazilian exports and anti-money laundering. In *2016 15th ieee international conference on machine learning and applications (icmla)*, pages 954–960. IEEE, 2016.

Manuel Peña-Fernández, Alejandro Serrano-Cases, Almudena Lindoso, Mario García-Valderas, Luis Entrena, Antonio Martínez-Álvarez, and Sergio Cuenca-Asensi. Dual-core lockstep enhanced with redundant multithread support and control-flow error detection. *Microelectronics Reliability*, 100:113447, 2019.

Ethan Perez, Florian Strub, Harm De Vries, Vincent Dumoulin, and Aaron Courville. Film: Visual reasoning with a general conditioning layer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Debjit Das Sarma Pete Bannon, Ganesh Venkataramanan. Fsd chip - tesla, 2019. URL https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip.

Vasileios Porpodas. Zofi: Zero-overhead fault injection tool for fast transient fault coverage analysis. *arXiv preprint arXiv:1906.09390*, 2019.

Yuxian Qiu, Jingwen Leng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. Adversarial defense through network profiling based path extraction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4777–4786, 2019.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. In *Advances in Neural Information Processing Systems*, pages 6076–6085, 2017.

Aditi Raghunathan, Sang Michael Xie, Fanny Yang, John C Duchi, and Percy Liang. Adversarial training can hurt generalization. *arXiv preprint arXiv:1906.06032*, 2019.

Parvathy Rajendran and Howard Smith. Implications of longitude and latitude on the size of solar-powered uav. *Energy conversion and management*, 98:107–114, 2015.

Sebastian Ramos, Stefan Gehrig, Peter Pinggera, Uwe Franke, and Carsten Rother. Detecting unexpected obstacles for self-driving cars: Fusing deep learning and geometric modeling. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1025–1032. IEEE, 2017.

Qing Rao and Jelena Frtunikj. Deep learning for self-driving cars: Chances and challenges. In *Proceedings of the 1st International Workshop on Software Engineering for AI in Autonomous Systems*, pages 35–38, 2018.

Andreas Reschka. Safety concept for autonomous vehicles. In *Autonomous Driving*, pages 473–496. Springer, 2016.

Anthony Thyron Rivers. *Modeling software reliability during non-operational testing*. North Carolina State University, 1998.

Francisca Rosique, Pedro J Navarro, Carlos Fernández, and Antonio Padilla. A systematic review of perception system and simulators for autonomous vehicles research. *Sensors*, 19(3):648, 2019.

Amir Roth. Store vulnerability window (svw): Re-execution filtering for enhanced load optimization. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 458–468. IEEE, 2005.

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.

Francisco Rubio, Francisco Valero, and Carlos Llopis-Albert. A review of mobile robots: Concepts, methods, theoretical framework, and applications. *International Journal of Advanced Robotic Systems*, 16(2):1729881419839596, 2019.

Yasir Dawood Salman, Ku Ruhana Ku-Mahamud, and Eiji Kamioka. Distance measurement for self-driving cars using stereo camera. In *International Conference on Computing and Informatics*, volume 1, pages 235–242, 2017.

Char Sample and Kim Schaffer. An overview of anomaly detection. *IT Professional*, 15(1):8–11, 2013.

Athena Sayles, Ashish Hooda, Mohit Gupta, Rahul Chatterjee, and Earlence Fernandes. Invisible perturbations: Physical adversarial examples exploiting the rolling shutter effect. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14666–14675, 2021.

Lukas Schroth. The drone market size 2020-2025: 5 key takeaways. https://droneii.com/the-drone-market-size-2020-2025-5-key-takeaways, 2020.

Ali Shafahi, Mahyar Najibi, Amin Ghiasi, Zheng Xu, John Dickerson, Christoph Studer, Larry S Davis, Gavin Taylor, and Tom Goldstein. Adversarial training for free! *arXiv preprint arXiv:1904.12843*, 2019.

Yiqiu Shen, Laura Heacock, Jonathan Elias, Keith D Hentel, Beatriu Reig, George Shih, and Linda Moy. Chatgpt and other large language models are double-edged swords, 2023.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Vasu Singla, Sahil Singla, Soheil Feizi, and David Jacobs. Low curvature activations reduce overfitting in adversarial training. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 16423–16433, 2021.

Jim Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

Michael D Smith. Overcoming the challenges to feedback-directed optimization (keynote talk). In *ACM SIGPLAN Notices*, volume 35, pages 1–11. ACM, 2000.

Daniel Sorin, Milo Martin, Mark Hill, and David Wood. Fast checkpoint/recovery to support kilo-instruction speculation and hardware fault tolerance. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2000.

Daniel J Sorin. Fault tolerant computer architecture. *Synthesis Lectures on Computer Architecture*, 4(1):1–104, 2009.

Vilas Sridharan and David R Kaeli. Quantifying software vulnerability. In *Proceedings of the 2008 Workshop on Radiation effects and fault tolerance in nanometer technologies*, pages 323–328, 2008.

Vilas Sridharan and David R Kaeli. Using hardware vulnerability factors to enhance avf analysis. *ACM SIGARCH Computer Architecture News*, 38(3):461–472, 2010.

Soumya Sudhakar, Sertac Karaman, and Vivienne Sze. Balancing actuation and computing energy in motion planning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4259–4265. IEEE, 2020.

Soumya Sudhakar, Vivienne Sze, and Sertac Karaman. Data centers on wheels: Emissions from computing onboard autonomous vehicles. *IEEE Micro*, 43(1):29–39, 2022.

Amr Suleiman, Zhengdong Zhang, Luca Carlone, Sertac Karaman, and Vivienne Sze. Navion: A fully integrated energy-efficient visual-inertial odometry accelerator for autonomous navigation of nano drones. In *2018 IEEE symposium on VLSI circuits*, pages 133–134. IEEE, 2018.

Amr Suleiman, Zhengdong Zhang, Luca Carlone, Sertac Karaman, and Vivienne Sze. Navion: A 2-mw fully integrated real-time visual-inertial odometry accelerator for autonomous navigation of nano drones. *IEEE Journal of Solid-State Circuits*, 54(4): 1106–1119, 2019.

Ke Sun, Zhanxing Zhu, and Zhouchen Lin. Towards understanding adversarial examples systematically: Exploring data size, task and model factors. *arXiv preprint arXiv:1902.11019*, 2019.

Yi Sun, Ding Liang, Xiaogang Wang, and Xiaoou Tang. Deepid3: Face recognition with very deep neural networks. *arXiv preprint arXiv:1502.00873*, 2015.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017a.

Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017b.

Takafumi Taketomi, Hideaki Uchiyama, and Sei Ikeda. Visual slam algorithms: A survey from 2010 to 2016. *IPSJ Transactions on Computer Vision and Applications*, 9(1):1–11, 2017.

Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, et al. Compute solution for tesla's full self-driving computer. *IEEE Micro*, 40(2): 25–35, 2020.

Max Taylor, Jayson Boubin, Haicheng Chen, Christopher Stewart, and Feng Qin. A study on software bugs in unmanned aircraft systems. In *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1439–1448. IEEE, 2021.

Dang Duy Thang and Toshihiro Matsui. Image transformation can make neural networks more robust against adversarial examples. *arXiv preprint arXiv:1901.03037*, 2019.

Yapeng Tian and Chenliang Xu. Can audio-visual integration strengthen robustness under multimodal attacks? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5601–5611, 2021.

Florian Tramer and Dan Boneh. Adversarial training and robustness for multiple perturbations. *arXiv preprint arXiv:1904.13000*, 2019.

Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017a.

Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017b.

Florian Tramer, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. On adaptive attacks to adversarial example defenses. *arXiv preprint arXiv:2002.08347*, 2020.

Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. Compiler optimization-space exploration. In *Proceedings of the international sym-*

posium on Code generation and optimization: feedback-directed and runtime optimization, pages 204–215. IEEE Computer Society, 2003.

Kyriakos G Vamvoudakis, Panos J Antsaklis, Warren E Dixon, João P Hespanha, Frank L Lewis, Hamidreza Modares, and Bahare Kiumarsi. Autonomy and machine intelligence in complex systems: A tutorial. In *2015 American Control Conference (ACC)*, pages 5062–5079. IEEE, 2015.

Sai Vemprala and Ashish Kapoor. Adversarial attacks on optimization based planners. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9943–9949. IEEE, 2021.

Micaela Verucchi, Luca Bartoli, Fabio Bagni, Francesco Gatti, Paolo Burgio, and Marko Bertogna. Real-time clustering and lidar-camera fusion on embedded platforms for self-driving cars. In *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, pages 398–405. IEEE, 2020.

Haotao Wang, Tianlong Chen, Shupeng Gui, Ting-Kuei Hu, Ji Liu, and Zhangyang Wang. Once-for-all adversarial training: In-situ tradeoff between robustness and accuracy for free. *arXiv preprint arXiv:2010.11828*, 2020a.

Xiangyu Wang, Xuyu Wang, Shiwen Mao, Jian Zhang, Senthilkumar CG Periaswamy, and Justin Patton. Adversarial deep learning for indoor localization. *IEEE Internet of Things Journal*, 2022.

Xingbin Wang, Rui Hou, Boyan Zhao, Fengkai Yuan, Jun Zhang, Dan Meng, and Xuehai Qian. Dnnguard: An elastic heterogeneous dnn accelerator architecture against adversarial attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 19–34, 2020b.

Xueyuan Wang and M Cenk Gursoy. Resilient path planning for uavs in data collection under adversarial attacks. *IEEE Transactions on Information Forensics and Security*, 2023.

Yulong Wang, Hang Su, Bo Zhang, and Xiaolin Hu. Interpret neural networks by identifying critical data routing paths. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018a.

Yulong Wang, Hang Su, Bo Zhang, and Xiaolin Hu. Interpret neural networks by identifying critical data routing paths. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8906–8914, 2018b.

Li-Hua Wen and Kang-Hyun Jo. Deep learning-based perception systems for autonomous driving: A comprehensive survey. *Neurocomputing*, 2022.

Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. Mitigating adversarial effects through randomization. *arXiv preprint arXiv:1711.01991*, 2017.

Cihang Xie, Mingxing Tan, Boqing Gong, Jiang Wang, Alan L Yuille, and Quoc V Le. Adversarial examples improve image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 819–828, 2020.

Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv preprint arXiv:1704.01155*, 2017.

Xiaogang Xu, Hengshuang Zhao, and Jiaya Jia. Dynamic divide-and-conquer adversarial training for robust semantic segmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7486–7495, 2021.

Xiaokun Yang, T Andrew Yang, and Lei Wu. An edge detection ip of low-cost system on chip for autonomous vehicles. In *Advances in Artificial Intelligence and Applied Cognitive Computing: Proceedings from ICAI'20 and ACC'20*, pages 775–786. Springer, 2021.

Yao-Yuan Yang, Cyrus Rashtchian, Hongyang Zhang, Ruslan Salakhutdinov, and Kamalika Chaudhuri. A closer look at accuracy vs. robustness. *arXiv preprint arXiv:2003.02460*, 2020.

Yulin Yang and Guoquan Huang. Map-based localization under adversarial attacks. In *Robotics Research: The 18th International Symposium ISRR*, pages 775–790. Springer, 2019.

Keun Soo Yim, Valentin Sidea, Zbigniew Kalbarczyk, Deming Chen, and Ravishankar K Iyer. A fault-tolerant programmable voter for software-based n-modular redundancy. In *2012 IEEE Aerospace Conference*, pages 1–20. IEEE, 2012.

Mingjun Yin, Shasha Li, Zikui Cai, Chengyu Song, M Salman Asif, Amit K Roy-Chowdhury, and Srikanth V Krishnamurthy. Exploiting multi-object relationships for detecting adversarial attacks in complex scenes. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7858–7867, 2021.

Jong-Hyeok Yoon and Arijit Raychowdhury. Neuroslam: A 65-nm 7.25-to-8.79-tops/w mixed-signal oscillator-based slam accelerator for edge robotics. *IEEE Journal of Solid-State Circuits*, 56(1):66–78, 2020.

Bo Yu, Wei Hu, Leimeng Xu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1067–1081. IEEE, 2020.

Kaiwen Yuan, Li Ding, Mazen Abdelfattah, and Z Jane Wang. Licas3: A simple lidar–camera self-supervised synchronization method. *IEEE Transactions on Robotics*, 38 (5):3203–3218, 2022.

Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 30(9):2805–2824, 2019.

Hongyang Zhang, Yaodong Yu, Jiantao Jiao, Eric Xing, Laurent El Ghaoui, and Michael Jordan. Theoretically principled trade-off between robustness and accuracy. In *International Conference on Machine Learning*, pages 7472–7482. PMLR, 2019.

Jingfeng Zhang, Jianing Zhu, Gang Niu, Bo Han, Masashi Sugiyama, and Mohan Kankanhalli. Geometry-aware instance-reweighted adversarial training. *arXiv preprint arXiv:2010.01736*, 2020a.

Jinlai Zhang, Lyujie Chen, Binbin Liu, Bo Ouyang, Qizhi Xie, Jihong Zhu, Weiming Li, and Yanmei Meng. 3d adversarial attacks beyond point cloud. *arXiv preprint arXiv:2104.12146*, 2021.

Min Zhang and Juntao Li. A commentary of gpt-3 in mit technology review 2021. *Fundamental Research*, 1(6):831–833, 2021.

Ming Zhang and Naresh R Shanbhag. Soft-error-rate-analysis (sera) methodology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):2140–2155, 2006.

Xiang Zhang and Marinka Zitnik. Gnnguard: Defending graph neural networks against adversarial attacks. *arXiv preprint arXiv:2006.08149*, 2020.

Zijie Zhang, Zeru Zhang, Yang Zhou, Yelong Shen, Ruoming Jin, and Dejing Dou. Adversarial attacks on deep graph matching. *Advances in Neural Information Processing Systems*, 33, 2020b.

Stephan Zheng, Yang Song, Thomas Leung, and Ian Goodfellow. Improving the robustness of deep neural networks via stability training. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 4480–4488, 2016.

Yuhao Zhu and Vijay Janapa Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24. IEEE, 2013.