Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Stian Jørstad Sulebak
Mikkel Svartveit

Extracting Instruction Set Characteristics from Raw Binary Code using Convolutional Neural Networks

Master's thesis in Computer Science
Supervisor: Donn Morrison
June 2025

NTNU
Norwegian University of
Science and Technology

Stian Jørstad Sulebak
Mikkel Svartveit

Extracting Instruction Set Characteristics from Raw Binary Code using Convolutional Neural Networks

Master's thesis in Computer Science
Supervisor: Donn Morrison
June 2025

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

This thesis investigates the application of Convolutional Neural Networks (CNNs) for detecting instruction set features from the binary code of a computer program, without explicit feature engineering. While prior research has focused on classifying the entire Instruction Set Architecture (ISA) of a binary file, we shift focus to detecting individual architectural properties that can generalize across unknown or undocumented instruction sets. We train and evaluate six CNN architectures of varying complexity for detecting endianness and fixed/variable instruction width, comparing simple models against deeper networks, one-dimensional versus two-dimensional convolutions, and models with and without embedding layers.

Using rigorous evaluation methods including Leave-One-Group-Out Cross-Validation (LOGO CV) and cross-dataset testing, we demonstrate that small CNN models with embedding layers can detect endianness with up to 90.3% accuracy and fixed/variable instruction width with up to 88.0% accuracy on unseen architectures within the ISAdetect dataset. However, we observe notable performance degradation when evaluating on more diverse datasets, with accuracy for unseen architectures dropping below 75% for both target features. This indicates generalization challenges when models encounter truly novel architectures.

We introduce BuildCross, a new dataset containing binary code from 40 different ISAs with an associated cross-compilation framework. Combining BuildCross with ISAdetect for training improves fixed/variable instruction width detection on the CpuRec dataset from approximately 55% to more than 80%, demonstrating that architectural diversity in training data is important for generalization.

Our results show that while CNNs can detect key ISA features from raw binary code with performance comparable to methods using manual feature engineering, their generalization capabilities are heavily dependent on training data diversity. This research contributes to advancing software reverse engineering capabilities, particularly for analyzing binaries from unknown or undocumented instruction set architectures.

Sammendrag

Denne oppgaven undersøker bruken av konvolusjonsnettverk (CNN) for å identifisere egenskaper ved instruksjonssettet til et dataprogram direkte fra binærkode, uten eksplisitt databearbeiding. Tidligere forskning har fokusert på å klassifisere hele instruksjonssettarkitekturen (ISA) til en binærfil. Vår tilnærming er i stedet å identifisere individuelle arkitekturegenskaper som kan generaliseres til ukjente eller udokumenterte instruksjonssett. Vi trener og evaluerer seks CNN-arkitekturer med varierende kompleksitet for å oppdage endianness (byterekkefølge) og fast/variabel instruksjonsbredde. Vi sammenligner enkle modeller med dypere nettverk, endimensjonale mot todimensjonale konvolusjoner, samt modeller med og uten embedding-lag.

Ved bruk av omfattende evalueringsmetoder som kryssvalidering med usett gruppe samt testing på tvers av datasett, viser vi at små CNN-modeller med embedding-lag kan oppdage endianness med opptil 90,3 % nøyaktighet og fast/variabel instruksjonsbredde med opptil 88,0 % nøyaktighet på usette arkitekturer i ISAdetect-datasettet. Vi observerer imidlertid en betydelig reduksjon i ytelse ved evaluering på mer varierte datasett, der nøyaktigheten for usette arkitekturer faller under 75 % for begge egenskapene. Dette tyder på generaliseringsutfordringer når modellene møter helt ukjente arkitekturer.

Vi introduserer BuildCross, et nytt datasett som inneholder binærkode fra 40 forskjellige ISAs med tilhørende rammeverk for krysskompilering. Ved å kombinere BuildCross med ISAdetect for trening forbedres deteksjonen av fast/variabel instruksjonsbredde på CpuRec-datasettet fra omtrent 55 % til over 80 %, noe som viser at arkitekturmangfold i treningsdataen er kritisk for generalisering.

Våre resultater viser at selv om CNNer kan oppdage sentrale ISA-egenskaper fra binærkode på nivå med metoder som benytter manuell databearbeiding, avhenger modellenes generaliseringsevne av mangfoldet i treningsdataen. Denne forskningen bidrar til å fremme ”reverse engineering” av programvare, spesielt for analyse av binærfiler fra ukjente eller udokumenterte instruksjonssettarkitekturer.

Contents

List of Figures

List of Tables

Acronyms

CISC Complex Instruction Set Computing

CNN Convolutional Neural Network

ELF Executable and Linkable Format

GCC GNU Compiler Collection

GPL GNU General Public License

GPU Graphics Processing Unit

HPC High Performance Computing

IoT Internet of Things

ISA Instruction Set Architecture

LOGO CV Leave-One-Group-Out Cross-Validation

MMCC Microsoft Malware Classification Challenge

NTNU the Norwegian University of Science and Technology

RISC Reduced Instruction Set Computing

p.p. percentage points

SGD United Nations Sustainable Development Goals

Chapter 1

Introduction

Software reverse engineering is the process of analyzing compiled binary programs to understand their functionality, structure, and behavior without access to the source code. Encountering compiled programs where the source code is unknown is common, particularly for proprietary software where the code is considered the intellectual property of the developing company. For these programs, reverse engineering can help third parties identify security vulnerabilities, detect malware, or ensure the quality of programs.

The software reverse engineering process is complex, and numerous techniques, frameworks, and tools exist for simplifying these tasks. A crucial step in most reverse engineering pipelines is code discovery and disassembly of the binary file [1]. To understand how a program works, one must first identify the location of the code section in the binary, and then break down the code section into individual machine instructions. To achieve this, knowledge of the Instruction Set Architecture (ISA) is critical. The ISA defines the contract between hardware and software, specifying how binary code should be executed on a given processor. Common ISAs include x86-64, ARM, RISC-V, and MIPS [2]. These instruction sets are well-documented, and disassembling binary programs is straightforward with readily available open-source tools.

However, not all CPUs are built on these common instruction sets. In particular, embedded systems require vastly different specifications than general-purpose computers, and often utilize custom ISAs that are completely undocumented and unknown to reverse engineers [3]. Additionally, certain programs are purposely difficult to reverse engineer thanks to virtualization obfuscation, a technique where programs are compiled for a randomized instruction set and executed through a corresponding custom virtual machine [4, 5]. With the rise of Internet of Things (IoT) and custom hardware, malware exploiting these new attack vectors presents a significant challenge for reverse engineers: when the specifications of the ISA are unknown, straightforward decompilation of the program binary is not feasible [6, 7]. Moreover, the methods for ISA detection outlined in prior literature rely on closed-set classification, that is, the binary can only be classified if its ISA is from a predefined list of architectures [8]. With programs compiled for embedded systems or custom virtual machines, this is often not the case, and we need alternative techniques to successfully reverse engineer these binaries.

Reverse engineers would benefit from an efficient and reliable way of discovering fundamental architectural properties from binary code when the specific ISA is unknown or lacks documenta-

tion. Uncovering architectural features such as endianness, word size, and instruction width is fundamental for reverse engineering programs of unknown ISAs [9].

Convolutional Neural Networks (CNNs) is a class of deep learning models for classifying unstructured, grid-based data. The use of convolution layers allows for deep networks that can capture sophisticated relationships without an excessive amount of computational power. Importantly, the nature of CNNs allows for automatically discovering significant patterns in the input data, without manually engineering features to train on. While CNNs are primarily used for processing and analyzing visual data like images, prior research has proven the utility of CNNs in various binary code analysis tasks, particularly within malware detection and classification [8]. We hypothesize that similar techniques can be used for detecting the ISA or uncovering specific architectural features from compiled binary code.

1.1 Objectives and research questions

This thesis investigates whether CNNs can be trained to detect individual ISA features from raw binary code, enabling the analysis of binaries from previously unseen architectures. Whereas prior research emphasizes ISA classification, we shift the focus to feature detection, training models to recognize fundamental architectural properties that can be generalized across ISA implementations. We leverage CNNs' ability to automatically and adaptively learn patterns from the input, eliminating the need for manual feature engineering.

The overarching research question that guides this thesis is:

> **RQ:** To what extent can CNNs effectively identify ISA features from raw binary programs without explicit feature engineering?

We break this down into three sub-questions:

> **RQ1**: Which ISA features can be classified with high accuracy by CNNs?
>
> **RQ2**: How does the choice of method for encoding software binaries impact the CNNs' ability to learn ISA characteristics?
>
> **RQ3**: How does the model architecture impact the CNN's ability to learn ISA characteristics?

1.2 Contributions

The main contribution of our work is a comprehensive evaluation of CNNs for detecting the ISA features endianness and fixed/variable instruction width. We train and evaluate six different CNN architectures, comparing the behavior of small and large models, as well as evaluating whether including embedding layers improves the classification performance. We implement comprehensive evaluation strategies, including Leave-One-Group-Out Cross-Validation (LOGO CV) and cross-dataset testing, to assess how our models perform on binaries from truly unseen ISAs. We analyze and compare our results to prior work that relies on feature engineering and traditional machine learning techniques, pointing out trade-offs in terms of accuracy, interpretability, data requirements, and computational resources.

Additionally, we contribute to the field of software reverse engineering by developing the BuildCross dataset and cross-compilation framework. This dataset is developed specifically for thor-

oughly testing and evaluating our proposed CNN models. It contains compiled binary code from 40 different ISAs, acquired by cross-compiling the source code of 9 widely-used open-source libraries. This results in roughly 120 MB of raw binary code. The dataset and associated cross-compilation framework are available on GitHub under an open-source GNU General Public License (GPL) [1].

1.3 Thesis structure

The remainder of this thesis is organized as follows.

Chapter 2 provides the theoretical foundation for our work. It includes background knowledge of low-level computer software fundamentals, the reverse engineering process, essential machine learning concepts, and an introduction to CNNs. We also review related work on ISA detection and CNN-based binary analysis.

Chapter 3 describes our experiments and their setup. We start by introducing the datasets, as well as the technical configuration and hyperparameters, used for training and testing our models. Then, we describe the development process for BuildCross, our custom dataset. Finally, we define our six CNN architectures, along with the evaluation strategies for measuring the performance of each of them.

Chapter 4 presents our findings for both endianness and fixed/variable instruction width classification, revealing model performance on both seen and unseen architectures.

Chapter 5 analyzes our results in depth. We summarize the key findings, dissect the behavior of each model, and assess their generalizability to unseen data. Then, we compare our approach to prior research. Moreover, we provide a quality assessment of the used datasets. Finally, we address some limitations of our approach, as well as briefly discuss the sustainability implications of our work.

Chapter 6 concludes our thesis, and summarizes how our findings answer the research questions. We also suggest areas for future research.

[1] https://github.com/mikkelsvartveit/thesis/releases

Chapter 2

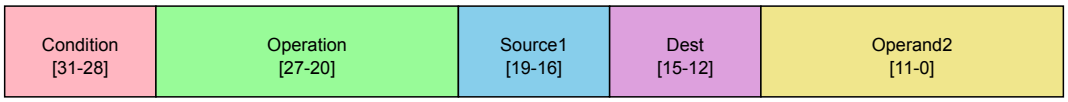
Background

This chapter presents the theoretical foundation required to understand the rest of the thesis. Starting with Section 2.1, we look at the basic, low-level concepts of computer software, instruction sets, and compilers. Section 2.2 then introduces software reverse engineering, which is the overarching topic of this thesis. Section 2.3 introduces the necessary concepts from the field of machine learning, with a focus on Convolutional Neural Networks (CNNs). Finally, Section 2.4 reviews some prior work related to the topic.

2.1 Computer software

2.1.1 Binary executables

All computer software boils down to a series of bytes readable by the CPU. The bytes are organized in *instructions*. An instruction always includes an *opcode* (Operation Code), which tells the CPU what operation should be executed. Depending on the opcode, the instruction often contains one or more *operands*, which provide the CPU with the data that should be operated on. The operands can be immediate values (values specified directly in the instruction), registers (a small, very fast memory located physically on the CPU), or memory addresses. Figure 2.1 illustrates the instruction format of ARM, which uses 32-bit instructions [10].



Example Instruction Types:

Data Processing (ADD R0, R1, R2)	COND | 00001000 | 0001 | 0000 | 000000000010
Memory Access (LDR R0, [R1])	COND | 01000001 | 0001 | 0000 | 000000000000
Branch (B label)	COND | 1010 | 24-bit offset

Figure 2.1: Instruction format and examples from the ARM instruction set [10].

2.1.2 Instruction set architectures

An Instruction Set Architecture (ISA) is a contract between hardware and software on how binary code should be run on a given computer. In the early days of computer programming, every new computer system was created with a new ISA, meaning programs had to be custom-written for each specific machine. IBM and their System/360 series, introduced in 1964, were the first to use the ISA as an abstraction layer between hardware and software. This new approach meant that despite having different internal architectures, all variations of the System/360 computers could run the same programs as they shared a common ISA [11]. The commercial success of this approach set an industry standard that continues to define modern computing, where hardware manufacturers can implement already established ISAs, ensuring cross-generational program compatibility [12].

In addition to defining an instruction set, the ISA gives a complete specification about how software interfaces with hardware, including how instructions can be combined, memory organization and addressing, supported data types, memory consistency models, and interrupt handling. Examples of well-known ISA families are x86, ARM, and RISC-V. Compilers can typically target multiple ISAs, allowing the same high-level source code to be executed on different architectures through appropriate translation to the target instruction set.

2.1.2.1 CISC and RISC

ISAs today generally fall into two camps: *Complex Instruction Set Computing (CISC)* and *Reduced Instruction Set Computing (RISC)*. CISC architectures, like x86, provide many specialized instructions that can perform complex operations in a single instruction. CISC can simplify complex operations at the programming level as well as potentially reduce code size but at the cost of requiring more complex hardware. RISC architectures, like ARM and RISC-V, favor simplicity with a smaller set of fixed-length instructions that execute in a single cycle, potentially making them more energy-efficient and easier to implement.

2.1.2.2 Instruction set

An important part of all ISAs is the instruction set, which defines the binary encoding of different instructions, providing a mapping of which bits and bytes translate to which instructions. Each instruction typically has a human-readable keyword (like ‘ADD’ or ‘MOV’), forming an assembly language that allows programmers to understand and write code at the machine level.

2.1.2.3 Word size

A fundamental characteristic of any ISA is its *word size*, which defines the natural unit of data the processor works with – typically 32 or 64 bits in modern architectures. This affects everything from register sizes to memory addressing capabilities. However, there is no standardized definition of what constitutes word size. One natural way to define it is the size of the registers in the CPU. For instance, this is why x86-64 is referred to as a 64-bit architecture. Even so, some software and documentation for the x86 family – such as the Microsoft Windows API – still define a single word to be 16 bits, even though the registers are 64 bits wide. Other definitions include the addressable memory space size, the data bus width, or the ALU input width, all of which may or may not be equal to the register size [13].

2.1.2.4 Endianness

The *endianness* determines how multi-byte values are stored in memory: *little-endian* architectures (like the x86 family) store the least significant byte first, while *big-endian* architectures store the most significant byte first. This is illustrated in Table 2.1. Endianness is a rudimentary characteristic of an ISA, but when analyzing and running a program, which order the bytes are stored in memory is crucial to understand. The endianness is typically determined by the ISA, but some architectures support both big-endian and little-endian modes. This is often referred to as *bi-endian* and is encountered in MIPS, PowerPC, and some modern versions of ARM [10, 14, 15]. Even though these architectures support endianness mode switching during runtime, programs are typically compiled for a single specific endianness, making the software binaries themselves either big-endian or little-endian. In other words, even though an ISA is classified as bi-endian, the endianness of a binary program is typically fixed at compile time.

(a) Big-endian

Address	0x1000	0x1001	0x1002	0x1003
Byte	0x12	0x34	0x56	0x78

(b) Little-endian

Address	0x1000	0x1001	0x1002	0x1003
Byte	0x78	0x56	0x34	0x12

Table 2.1: Comparison of how a 32-bit integer is stored in big-endian and little-endian.

2.1.2.5 Instruction width

The *instruction width* refers to the size, typically measured in bits, of a single CPU instruction. Some architectures, such as ARM64, have *fixed-width instructions*, where each instruction has the same size. Others, such as most CISC instruction sets, have *variable-width instructions*, where the size of each instruction can vary based on factors such as the opcode and the addressing mode. For instance, x86-64 programs can contain instructions ranging from 8 to 120 bits. A comparison between fixed width and variable width instruction sets is displayed in Figure 2.2. The degree of variability in instruction widths differs: some architectures, such as x86, support a wide range of sizes. Others, such as Blackfin and Epiphany, are limited to specific combinations, where instructions are either 16 bits or 32 bits. Instruction width has implications for binary program alignment in terms of analysis. Binaries from fixed-width ISAs are, in principle, easier to analyze since instructions are consistently aligned to specific byte boundaries, while variable-width ISAs complicate the identification of the beginning and end of instructions.

2.1.3 Compilers

Software developers employ tools like compilers and interpreters to convert programs from human-readable programming languages to executable machine code. In the very early days of computer programming, software had to be written in assembly languages that mapped instructions directly to binary code for execution. Growing hardware capabilities allowed for more complex applications, however, the lack of human readability of assembly languages made software increasingly difficult and expensive to maintain. In order to overcome this challenge, compilers were created to translate human-readable higher-level languages into executable programs. In the early 1950s, there were successful attempts at translating symbolically heavy

x86-64 Architecture

```
4156            pushq %r14
4155            pushq %r13
4154            pushq %r12
55              pushq %rbp
53              pushq %rbx
4883ec60        subq  $0x60, %rsp
4889442458      movq  %rax, 0x58(%rsp)
31c0            xorl  %eax, %eax
83ff01          cmpl  $0x1, %edi
0f8f32020000    jg    0x400d87 <.text+0x257>
488d6c2454      leaq  0x54(%rsp), %rbp
4c8d64243c      leaq  0x3c(%rsp), %r12
41be01000000    movl  $0x1, %r14d
0f1f8000000000  nopl  (%rax)
488d5c2430      leaq  0x30(%rsp), %rbx
```

ARM64 (AArch64) Architecture

```
d14007ff    sub   sp, sp, #0x1, lsl #12
d11843ff    sub   sp, sp, #0x610
a9b67bfd    stp   x29, x30, [sp, #-0xa0]!
910003fd    mov   x29, sp
a90363f7    stp   x23, x24, [sp, #0x30]
6d0627e8    stp   d8, d9, [sp, #0x60]
2a0003f8    mov   w24, w0
f0000080    adrp  x0, 0x414000
91066000    add   x0, x0, #0x198
6d072fea    stp   d10, d11, [sp, #0x70]
a90573fb    stp   x27, x28, [sp, #0x50]
6d0837ec    stp   d12, d13, [sp, #0x80]
aa0103fc    mov   x28, x1
f9400001    ldr   x1, [x0]
f90b57a1    str   x1, [x29, #0x16a8]
```

Figure 2.2: Comparison of disassembly of binary programs between x86-64 and ARM64 architecture. The binary encoding of the instructions on the left side of the assembly illustrates the variable width of x86-64 instructions compared to the fixed width of ARM64 instructions.

mathematical language to machine code. The language FORTRAN, developed at IBM in 1957, is generally considered the first complete compiled language, achieving efficiency near that of hand-coded applications. While languages like FORTRAN were primarily used for scientific computing needs, the growing complexity of software applications drove the development of more advanced operating systems and compilers [16]. One such advancement was the creation of the C programming language and its compiler in the early 1970s. Modern compilers (like GNU Compiler Collection (GCC) and Clang) can analyze the semantic meaning of the program, usually through some form of intermediate representation [17]. The ISA of the target system provides the compiler with the recipe to translate the intermediate representation into executable code. The intermediate representation is usually language- and system architecture-agnostic, which allows a compiler to translate the same program to many computer architectures.

The evolution of compilers brought significant advantages in code portability and development efficiency. Programming languages' increasing abstraction away from machine code was necessary to achieve efficient development and portability across different computer architectures [17]. However, this combined with other transformations done by compilers increasingly widened the gap between the source code and the binary executable. By separating the program's logic from its hardware-specific implementation, developers could write code once, compile, and run it on every platform they wanted, at the cost of making it more difficult to understand what a binary program does.

2.1.4 Embedded targets and cross-compilation

Embedded systems are specialized computing devices integrated within larger systems to perform specific and dedicated functions. Thus, unlike general-purpose computers, embedded systems designed for specific tasks are typically optimized for reliability, power efficiency, and cost-effectiveness. Embedded systems power everything from household appliances like refrigerators and washing machines to networking equipment like routers, and a vast array of Internet of Things (IoT) devices. Most embedded systems are characterized by resource constraints, including limited memory, processing power, and energy capacity. In order for these systems to perform in such environments, embedded platforms typically incorporate

specialized hardware with custom processors and peripherals optimized for specific tasks. As a result, many embedded systems feature limited or no user interface and can only be controlled programmatically.

Since these specialized systems frequently use custom hardware with ISAs different from standard desktop or server computers, they present unique challenges for software development. Unlike general-purpose computers, which are often used to create programs for the same platform they are built on, it's often impractical or impossible to compile code directly on the target device. Developers typically have to use a technique called *cross-compilation* to build software for embedded systems.

2.1.4.1 Cross-compilation

Cross-compilation is the process of generating executable code for a platform (target) different from the one running the compiler (host). This approach allows developers to use more powerful development systems to create software for resource-constrained target devices.

In cross-compilation terminology, three distinct systems are involved in the process. The host system is the computer system where the compilation of programs occurs, providing the computational resources needed for the build process. The target system is where the compiled code will eventually run, often with limited resources that would make direct compilation impractical. The build system refers to the system where the compiler itself was built, which is the same as the host system in most development scenarios [18, 19].

A *cross-compiler toolchain* is the collection of software tools necessary to build executables for a target system. A complete toolchain consists of several key components working together. The compiler, such as GCC or Clang, serves as the core tool that converts source code to machine code appropriate for the target architecture. Binary utilities (like Binutils) provide essential tools for creating and managing binary files across different architectures. The C/C++ standard library supplies standard functions and data structures optimized for the target system, while a debugger helps identify and fix issues in the compiled program, often supporting remote debugging capabilities for target hardware [19, 20].

2.1.4.2 GNU Compiler Collection and GNU Binutils

The GNU Compiler Collection (GCC) is a comprehensive compiler system supporting various programming languages including C, C++, and Fortran. GCC started as the GNU C Compiler in 1987 but was later renamed as it expanded to support more languages. GCC is designed to be highly portable and can be built to run on various operating systems and hardware architectures. It features a modular design, using internal intermediate representations that are largely agnostic to the host and target systems [20, 21]. This lets much of the optimization logic and transformation work with different front-ends for different programming languages and different back-ends to generate code for a wide range of ISAs. GCC by itself takes in an input file in supported languages like C and outputs assembly for the target architecture. Each instance of a GCC compiler is configured to target a specific architecture, and this flexibility allows developers to compile versions of GCC to build software for different platforms.

GCC is not able to create working executables by itself. However, behind the scenes, GCC sets up a pipeline consisting of different tools to create executable programs. A common pairing to set up this pipeline with GCC is GNU Binutils, which is a collection of binary program tools that are

designed to work alongside compilers to create and manage executables. Some key components of Binutils include:

- **as**: The GNU assembler, which converts assembly language to machine code
- **ld**: The GNU linker, which combines machine code files into executables or libraries
- **ar**: Creates, modifies, and extracts from archive files (static libraries)
- **objcopy**: Copies and translates object files between formats
- **objdump**: Displays information about object files, including disassembly
- **readelf**: Displays information about Executable and Linkable Format (ELF) files

When invoking GCC to create a final executable, the compiler automatically calls the appropriate Binutils tools to create the final executable, and an illustration of this pipeline can be seen in Figure 2.3. Since the final executable depends on the target system, the GCC compiler and Binutils tools must be configured to target the same architecture. This is typically done by specifying the target architecture when building the toolchain, after which the GCC compiler will use the appropriate Binutils tools to generate the final executable for that architecture [[22]; [19];].

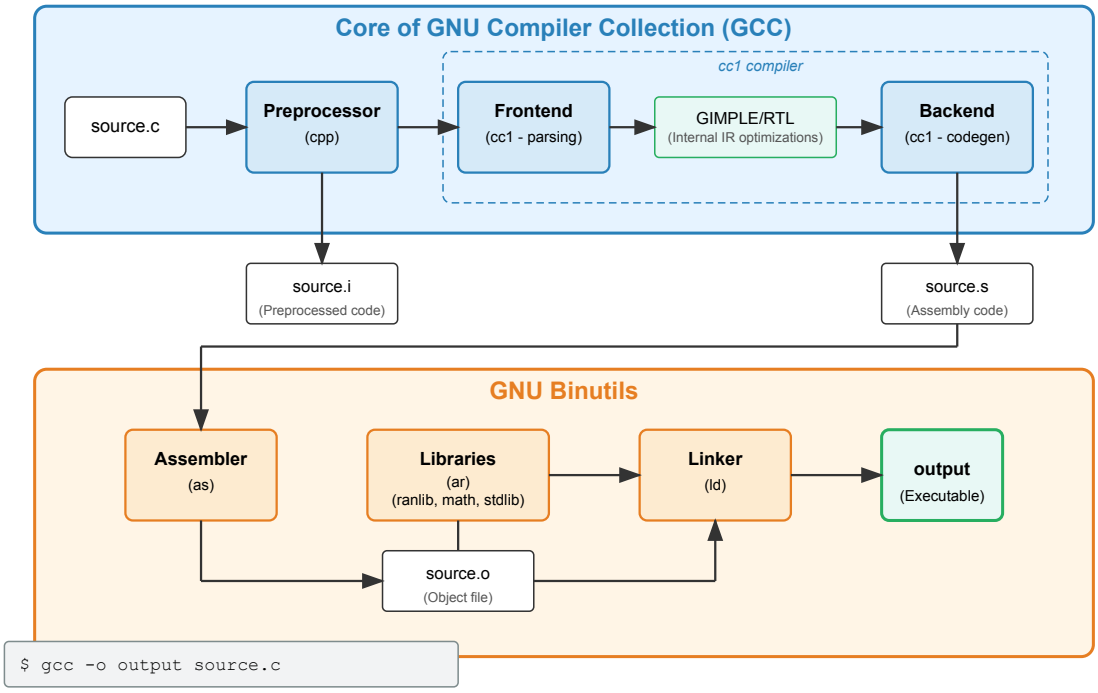


Figure 2.3: Illustration of how the GCC with Binutils pipeline compiles a source program, source.c, into an output executable.

All of the GNU projects are distributed under the GNU General Public License (GPL), which allows users to freely use, modify, and distribute the software. This has made GCC and Binutils widely adopted in open-source projects and embedded systems development.

2.1.4.3 Binary file formats and structures

Binary files come in different formats, each with its own structure and purpose, and understanding these formats is necessary for working with compiled code. In order for a program to be interpreted or executed by a computer it must be stored in a way that the CPU can read and understand. One way of doing this is to include a header at the beginning of the file that contains metadata about the executable. While different file formats exist, like Portable Executable for

Windows and Mach Object for macOS, the most common binary file format for Unix-like systems and many embedded devices is the Executable and Linkable Format (ELF). Since ELF is the most commonly seen file format in cross-compilation environments and embedded systems, it is the focus of this section.

ELF is a flexible and extensible binary file format that can be used for executables, object code, shared libraries et cetera. The ELF header contains information about the file type, ISA, entry point address for where to start execution, and section headers. It also includes support for debugging information, symbol tables, and relocation entries, making it easier to analyze and debug programs.

ELF files are organized into so-called sections, which are contiguous blocks of information within the binary file serving different purposes. The section names are conventionally prefixed with a period, and common sections in ELF files include:

- **.text**: Contains the executable code (machine instructions). Often referred to as the code section.
- **.data**: Contains initialized global variables
- **.bss**: Contains space for uninitialized global variables
- **.rodata**: Contains read-only data like strings
- **.symtab**: Symbol table with function and variable names and their locations

File formats like ELF provide a standardized way to represent the structure of a binary file, including sections for code, data, and metadata. When cross-compiling using the GCC/Binutils suite, ELF is often the default file format when targeting embedded targets. Therefore, ELF headers are typically found in binary files that serve different purposes in the creation of an executable program:

- **Object files**: Compiled source code files that contain machine code but are not yet executable. They must be linked together to create complete programs, as illustrated in Figure 2.3.
- **Executable files**: The final output of the compilation process that can be directly run by the operating system. These files contain the complete program with all dependencies resolved by the linker.
- **Archive files/Static libraries**: Static libraries are bundled as archive files, which are collections of object files grouped for easy reuse and distribution. Archive files have a global header with additional metadata, followed by the ELF-formatted object files. When a program is linked against a static library, the necessary object code is copied into the final executable.

2.2 Software reverse engineering

Software reverse engineering is a systematic process of analyzing and understanding how a program works without access to its source code or internal documentation. At its core, reverse engineering involves working backward from a compiled program to comprehend its functionality, architecture, and behavior – the opposite direction of traditional software development. Reverse engineering has its origins in hardware reverse engineering, where analysis of competitors' designs was used to gain a competitive advantage. Today, software reverse engineering is primarily

used for understanding program behavior, not for replicating it. Whether investigating potentially malicious code or maintenance of legacy systems, software reverse engineering provides insights when source code and documentation are unavailable [23, 24, 25, 26].

Software reverse engineering serves many purposes in today's digital world. In the domain of cybersecurity, it enables many types of vulnerability detection, where security researchers and bug hunters identify exploitable pieces of code. It can also be used to identify and analyze malware, protecting critical systems from infected executables, and preventing cyberattacks [26, 27, 28, 29]. Beyond cybersecurity, reverse engineering enables software interoperability by allowing engineers to understand how systems interact when documentation is unavailable. It can play a vital role in software maintenance, especially for legacy systems where original documentation or development expertise has been lost. Software reverse engineering also serves important legal and compliance functions, helping organizations verify adherence to security standards and licensing requirements. It can also support digital forensics through code similarity detection and ownership attribution [24, 25, 26, 29, 30, 31, 32].

While software reverse engineering encompasses a broad range of activities beyond the scope of this thesis, understanding the complete process provides context for our research contribution. Although we focus specifically on binary reverse engineering at the lowest level, presenting more of the reverse engineering workflow helps position our work within the larger field. The reverse engineering landscape can be viewed through two perspectives: the cognitive strategies and approaches reverse engineers employ when analyzing programs, and the practical tools and transformations they use to facilitate this analysis. In the following sections, we explore both the methodical process reverse engineers follow and some of the tools that enable their work at different levels of code abstraction.

2.2.1 The reverse engineering process

The thought process of a reverse engineer is often iterative and exploratory, as they try to understand the program's behavior and functionality [25, 26]. Votipka et al. conducted a survey of reverse engineers and found that the most common high-level steps in the reverse engineering process can be grouped into three phases [29]:

1. **Overview**
   The reverse engineer tries to establish a high-level understanding of the program. Some reverse engineers report that programs subject to analysis come with some information, which helps point the analysis in the right direction. A common strategy is to list strings used by the programs, often available in clear text in the data section of an executable. These strings often give hints about the domain and environment and might also point to external API calls. They also look at loaded resources and libraries and try to identify important functions and code segments.

2. **Subcomponent Scanning**
   In this phase, the reverse engineer scans through prioritized functions and code sections identified in the Overview step. In this scan the reverse engineer looks for so-called beacons; important nuggets of information like API calls, strings, leftover symbol names from compilation, control-flow structures, et cetera. If a function outline is found, the reverse engineer will try to identify the input and output of the function, and how it interacts with

other subroutines. Some common algorithms, loops, and data structures can be recognized from experience, and marked for future analysis.

3. **Focused Experimentation**
   Here, the reverse engineer tests specific hypotheses through program execution. This can be done by running the program in a debugger to examine memory states, manipulating the execution environment (like registry values or binary patching), comparing to known implementations of suspected algorithms, and, when necessary, reading code line-by-line for detailed understanding. Another valuable strategy is fuzzing: varying inputs to subcomponents and testing against expected changes to the output. The results from this phase are then fed back into the subcomponent scanning phase for iterative refinement.

Reverse engineers adapt their approach depending on the problem. The strategies used often come from experience and intuition, and the reverse engineer has to adapt their strategy to the specific program. This makes it difficult to create a one-size-fits-all approach to reverse engineering. The process is often not linear, and the frequently reverse engineer jumps between phases and tasks [25, 29]. However, a key theme across the sources is the need to separate the program into bite-sized blocks, hypothesize how they work, and test that hypothesis with experimentation. Another important strategy is to visualize the program flow, like drawing out the control flow graph or how different subcomponents connect. This helps the reverse engineer understand how different components interact and how data flows through the program [25, 26].

2.2.2 Tools and challenges

In order for reverse engineers to analyze a program, the code needs to be in a human-readable format. Software reverse engineering is reliant on tools that transform programs into digestible forms, like binary code to assembly, or assembly to source languages like C. Each level of abstraction comes with unique challenges that might need to be overcome in order to apply the tools and reverse engineer the program.

2.2.2.1 Binary reverse engineering and disassemblers

At the lowest level, when presented with a binary of unknown origin, reverse engineers use *disassemblers* like objdump, angr, and IDA Pro along with obtained knowledge of the ISA to translate the binary into assembly instructions [33, 34, 35]. Metadata about the ISA and target system is usually present in binary file headers like ELF, making disassembly quite simple for known architectures. Some of these tools are also able to recognize the ISA from a closed set of known architectures based on the binary code directly.

The primary challenge in binary disassembly arises when the target uses an unknown or undocumented ISA. Without existing disassembler support, reverse engineers must discover instruction encoding through statistical analysis and pattern recognition, which is a significantly more complex task. However, this approach is only feasible by knowing fundamental architectural properties first, such as endianness, instruction encoding format and width, word size, and whether the architecture is stack-based or register-based. Chernov & Troshina demonstrate this process systematically, beginning with statistical frequency analysis to identify return instructions, then using correlation patterns to discover call and jump instructions, before progressing to arithmetic and memory operations [9]. In addition to instruction identification, unknown ISAs present additional challenges in determining structural elements such as code sections, program entry points,

and function boundaries, which are all essential for meaningful execution flow analysis. Binaries can also be compressed or encrypted, all of which inhibits disassembly and reverse engineering [26, 35, 36, 37].

2.2.2.2 Decompilers and higher-level analysis

While assembly captures the semantic meaning of a program in a more human-readable manner and smaller pieces of code can be analyzed by reverse engineers, larger software systems are often too complex for meaningful information to be extracted purely through disassembly. *Decompilers* are one such tool to aid in obtaining higher-level understanding of the software. Decompilers like IDA Pro, angr, and Ghidra use assembly to reconstruct the program in a higher-level language like C to improve human readability and make program semantics easier to understand [33, 34, 38]. However, inherent limitations with information loss during compilation make it virtually impossible to reconstruct the source code from assembly. Software developers rely on variable names and code comments to document data structures and code, which are often lost during release builds of the software. Modern compilers also perform performance or memory optimizations like loop unrolling, function inlining, changing arithmetic operands, and control flow optimizations that can significantly transform the original code structure, making it even more challenging to map between source code and the resulting assembly [26, 29, 35].

In addition to disassembly, some tools can lift binaries into higher-level representations like language-invariant intermediate representations (such as LLVM IR) through semantic analysis. BinJuice is one such semantic analysis tool, which tries to capture program state changes performed by code blocks [39]. While intermediate representations are typically used as a stepping stone in compilation and decompilation, their language-agnostic nature makes them valuable for large-scale program analysis. By also comparing code at the intermediate representation level, analysts enhance code similarity detection and semantic analysis, which can help enable understanding of program behavior and structure [26, 35].

2.2.2.3 Obfuscation

Obfuscation is a technique that aims to make reverse engineering more difficult, by transforming the code in a way that preserves its functionality but makes it harder to understand. This can be achieved through various methods, such as manipulating the control flow of the program in order to make execution harder to follow, altering common data structure layouts and strings, and changing the layout of the file itself and the order of instructions. Tools like Tigress and Obfuscator-LLVM can take in a working program and apply these transformations automatically [40, 41].

Another advanced obfuscation technique involves creating custom virtual machines that execute programs using proprietary or random instruction sets incompatible with standard CPUs. This process requires two compilation steps: first, the original program is compiled to target the custom virtual machine's instruction set, and then the virtual machine itself is compiled for the host platform. The host system can then execute the obfuscated code only through this intermediary virtual machine layer, essentially letting a CPU execute code from any arbitrarily constructed ISA. These virtual machines are sometimes referred to as emulators or interpretes, and execution of the virtualized program is similar to the execution pipeline of interpreted and bytecode-based languages like Python, Java, and JavaScript. The custom instruction set makes it very difficult to disassemble the program, as the custom ISA based executed code can be constructed in ways that

do not match the binary encoding of other known architectures. Static analysis of the program is often impossible without first reverse engineering the custom virtual machine or by reverse engineering the custom instruction set from scratch like described in Section 2.2.2.1 [4, 5, 9].

Obfuscation serves multiple purposes: protecting intellectual property, hindering bug discovery and exploitation, and generally deterring reverse engineering efforts. However, it can also be employed maliciously to evade malware detection systems. While obfuscation complicates reverse engineering, the underlying program semantics remain unchanged, meaning that the obfuscated code is functionally equivalent to the original. This semantic equivalence ensures that with enough effort and appropriate techniques, reverse engineers can still decipher the program's behavior and functionality [4, 5, 27, 31, 32].

2.3 Machine learning

2.3.1 Deep learning

Modern machine learning has roots all the way back to the 1950s, when the first artificial neural network was implemented. The term machine learning was introduced around the same time. The development of neural networks continued during the 60s and 70s alongside statistical learning methods such as the nearest neighbor and decision tree methods. However, advances in neural networks faced challenges due to research that demonstrated fundamental limitations of single-layer networks. In the 1980s, the backpropagation algorithm (which is still widely used today) was popularized and solved the problem of training multi-layer networks. This put neural networks back on the map, and the field continued advancing through the 90s and 2000s. Significant breakthroughs such as parallelized training pipelines, generational leaps in computing power, and utilization of Graphics Processing Units (GPUs) for machine learning tasks accelerated deep learning developments. This led up to a turning point in 2012, when AlexNet, a deep convolutional neural network, won the annual ImageNet competition. Since then, deep learning approaches have dominated the field of machine learning.

2.3.2 Convolutional neural networks

CNNs is a deep learning technique designed for processing grid-based data. It is most commonly applied to visual tasks such as image classification and object detection. The main invention of CNNs is the concept of convolution layers. These layers scan across the input using *kernels*. The kernels detect features such as edges, textures, and patterns in the input data, and each outputs a feature map that is passed to the next layer. Each kernel has parameters that are trained based on the entire input grid. Figure 2.4 shows an example of a kernel sliding over the input.

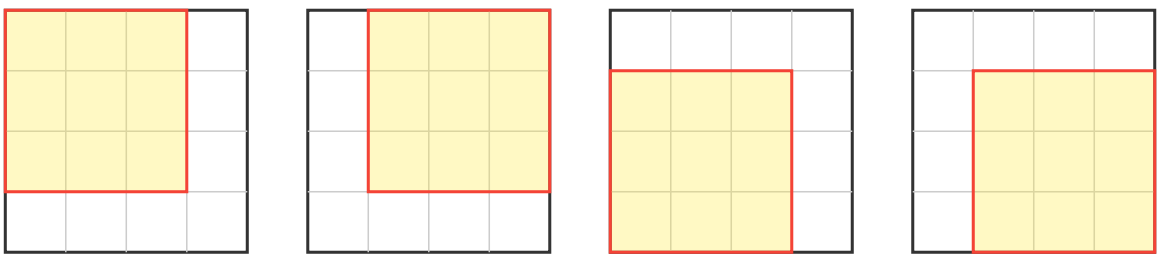


Figure 2.4: A 3x3 kernel sliding over a 4x4 input. This layer will result in a 2x2 output.

Most CNN architectures also use pooling layers, which are static, non-trainable layers that reduce the spatial dimensions of the data. Activation layers, usually ReLU, are used to introduce non-linearity into the network. Finally, fully-connected layers at the end of the network are used for final classification. Figure 2.5 shows an example of a basic CNN architecture.

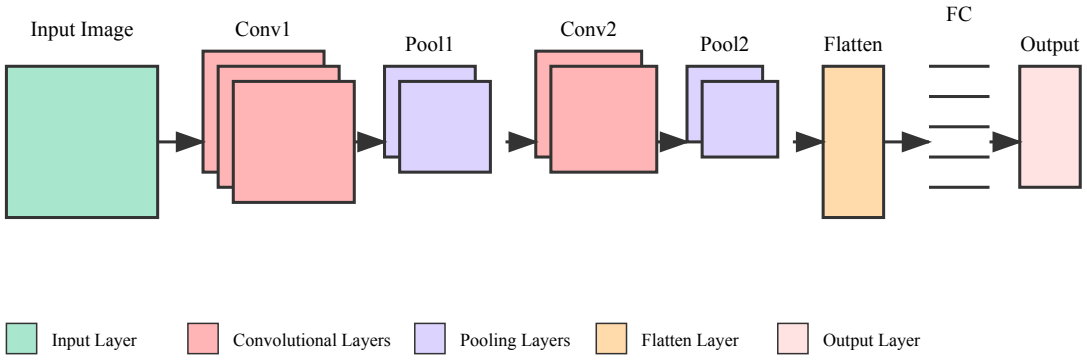


Figure 2.5: Simple CNN architecture

CNNs provide several advantages over competing approaches:

- Where traditional computer vision methods usually require significant feature engineering efforts, a CNN is able to automatically detect and learn features from the input data without manual feature extraction. This saves time and effort, and even enables models to detect patterns that human intuition would be unable to.
- CNN is more computationally efficient than fully connected neural networks. Where fully connected networks need parameters for every single connection between neurons, a CNN uses the same kernels across the entire input, which dramatically reduces the number of trainable parameters. Additionally, the nature of CNNs makes them more feasible for parallelization, better utilizing specialized hardware such as GPUs.
- CNN models are *translation invariant*. This means that they can recognize objects, patterns, and textures regardless of their spatial position in the input. This makes the models more versatile and generalizable than fully connected neural networks.

Hundreds of different CNN architectures have been proposed in previous literature. LeNet-5, which is considered the first modern CNN architecture, has around 60,000 trainable parameters [42]. Today, large-scale CNN architectures such as VGG-16 often have over 100 million trainable parameters [43].

Choosing a CNN architecture is often a trade-off between several factors:

- **Dataset size**: In general, more complex models require larger datasets. In cases where training data is limited, smaller architectures should be considered. Small dataset sizes combined with complex networks often lead to overfitting, meaning the model matches the training data so well that it fails to generalize to unseen data.

- **Training resources:** Larger models are more expensive to train in terms of computation power. Training deep learning models efficiently often requires the use of powerful GPUs.
- **Inference resources:** Larger models do not only increase the cost of training, they also increase the cost of inference, i.e. making predictions using the trained model. Depending on where the model will be deployed, this may be a deciding factor.

2.3.3 Overfitting and regularization

When training machine learning models, especially when model complexity is high, there is a risk of overfitting. Overfitting occurs when a model learns the training data too perfectly, including its noise and random fluctuations, rather than learning the true underlying patterns. A model that is overfitting to the training data will fail to generalize to unseen data, which causes weak performance in real-world applications.

Figure 2.6 shows an example of performance behavior of a model that is overfitting. We see that both training accuracy and validation accuracy improve initially, but after around 8 epochs, the validation accuracy stagnates while training accuracy keeps increasing.

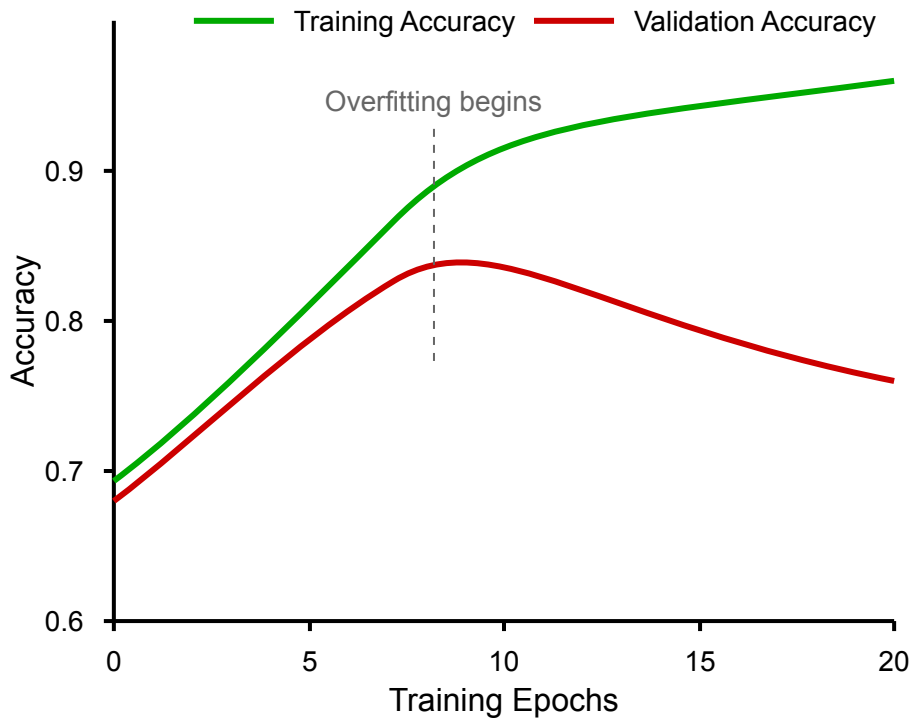


Figure 2.6: Example of training and validation accuracy when overfitting

Regularization is a set of techniques for reducing overfitting in machine learning models. In general, regularization penalizes models that are very complex, which incentivizes simpler models that likely generalize better to unseen data.

2.3.3.1 Ridge and Lasso regularization

Lasso (L1) regularization and Ridge (L2) regularization both work by appending a regularization term to the loss function. The standard loss function for Mean Squared Error (MSE) is:

$$L_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Lasso (L1) regularization sums up the absolute value of all model weights, multiplies it by a regularization strength λ, and adds this term to the loss function:

$$L_{lasso} = \underbrace{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}_{\text{MSE}} + \underbrace{\lambda \sum_{j=1}^{p} |w_j|}_{\text{L1 penalty}}$$

Ridge (L2) regularization sums up the squared value of all model weights, multiplies it by a regularization strength λ, and adds this term to the loss function:

$$L_{ridge} = \underbrace{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}_{\text{MSE}} + \underbrace{\lambda \sum_{j=1}^{p} w_j^2}_{\text{L2 penalty}}$$

Since the training process tries to update the model parameters in such a way that the loss function is minimized, both these regularization techniques result in models that balance fitting on the training data with maintaining simplicity.

2.3.3.2 Weight decay

Where L1 and L2 regularization add a regularization term to the loss function, weight decay instead modifies the weight update rule to include a decay factor. The standard weight update rule for gradient descent is:

$$w_t = w_{t-1} - \eta \frac{\partial L}{\partial w}$$

Weight decay scales down the previous parameter value by a factor of $(1 - \eta\lambda)$, where η is the learning rate and λ is the weight decay coefficient:

$$w_t = w_{t-1} \underbrace{(1 - \eta\lambda)}_{\text{Decay factor}} - \eta \frac{\partial L}{\partial w}$$

This causes the weights to gradually decay unless the gradient update is large enough to counteract it, leading to simpler models with smaller weights.

When using standard gradient descent for training, weight decay is mathematically equivalent to Ridge (L2) regularization. However, with adaptive optimizers such as Adam, L2 regularization gets scaled by the adaptive learning rates. Weight decay avoids this issue by applying the penalty directly to the weights, which makes it preferable to L2 regularization for these optimizers.

2.3.3.3 Dropout

Dropout is a regularization technique that randomly deactivates a subset of neurons during each training iteration. During training, each neuron has a probability p of being temporarily removed

from the network along with its connections. For the next iteration, all neurons are restored before randomly dropping a new subset.

During testing and inference, no neurons are dropped. Instead, all neuron outputs are scaled by p to maintain the same expected magnitude of activations:

$$h_{\text{test}} = p \cdot h$$

By forcing the network to function without access to all neurons during training, dropout prevents any neuron from relying too heavily on a specific subset of other neurons. This encourages the network to learn redundant representations, which improves robustness. The technique is particularly effective for very deep networks, where common dropout probabilities range from 0.2 to 0.5.

2.3.4 Cross-validation

Cross-validation is a technique used to assess the performance and generalizability of a machine learning model. It involves partitioning data into subsets, where the model is trained on certain subsets while validated using the remaining ones. The process is repeated, making sure the model is trained and validated using different splits. This helps reduce overfitting on a fixed validation set, with the trade-off of requiring more computation since the model needs to be trained multiple times.

It is worth noting that cross-validation is used only when verifying the model architecture and hyperparameters, not when training the actual model that will be deployed. After performance is assessed using cross-validation techniques, the final model is trained on all available training data without holding out a validation set.

2.3.4.1 K-fold cross-validation

In K-fold cross-validation, the data is randomly split into K equal-sized subsets called *folds*. Then, the model is trained K times, once for each fold. In each run, one fold is used for validating the model performance, while the remaining $K - 1$ folds are used as the training data. The model performance is typically aggregated across the runs to provide metrics for overall cross-validated performance. Figure 2.7 illustrates how the data can be split in a 5-fold cross-validation. Each fold serves as validation data once while the remaining data is used for training.

2.3.4.2 Leave-one-group-out cross-validation

Leave-One-Group-Out Cross-Validation (LOGO CV) is a variation used in cases where the data is naturally grouped into distinct clusters or categories. The purpose is to ensure that the trained model is tested on data independent of the groups it was trained on. Instead of partitioning the data into random subsets, it is split into groups based on a predefined variable. For each iteration, one group is left out as the validation set, and the model is trained on the remaining groups. This technique assesses how well the model generalizes to completely unseen groups, which is especially useful in case the final model will be used with data that does not belong to any of the groups present in the training data.

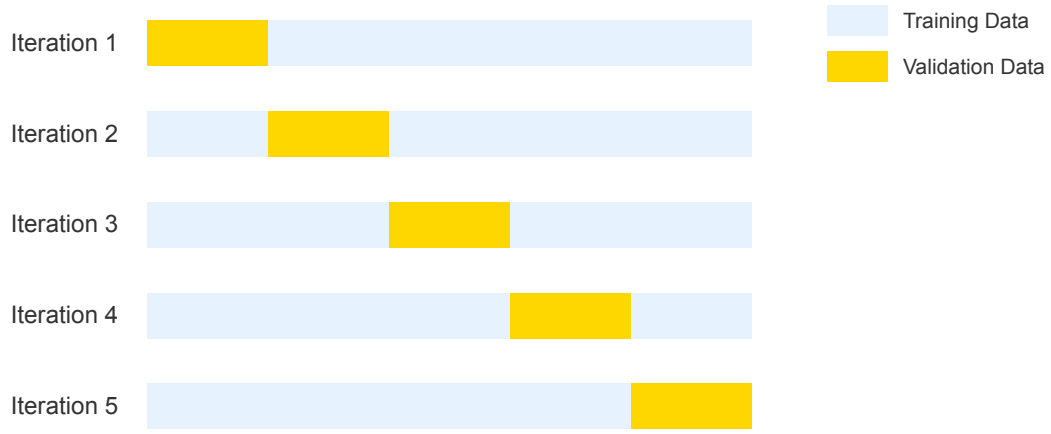


Figure 2.7: K-fold cross-validation with 5 folds.

2.3.5 Embeddings

Embeddings are a way to convert discrete data like words, categories, or items into continuous vectors of numbers that capture semantic relationships or patterns. These vectors are part of the model's trainable parameters, which allows the model to discover semantics from categorical input data during training. It is common to include an embedding layer as the first layer of a deep learning model, essentially creating a mathematical representation of arbitrary categorical data.

Word embeddings serve as the foundation for many natural language processing tasks. When trained on massive English datasets, these embeddings often capture sophisticated semantic relationships. A classic example in the literature demonstrates this through vector arithmetic: starting with the vector of "king", subtracting "man", and adding "woman", we end up with a vector that is very close to that of "queen" [44].

2.3.6 Statistical evaluation and hypothesis testing

Statistical evaluation is a crucial part of machine learning, as it helps us understand and draw conclusions on how well our model performs. Due to the stochastic nature of machine learning, since with many models the training process involves random initialization and sampling, it is important to use statistical methods to ensure that our results are not just due to chance. This is especially important when comparing different models or hyperparameters, as we want to be able to conclude whether the differences in performance are statistically significant. In this thesis, we tackle binary classification problems, where the model predicts which of two classes the input belongs to. In this section, we will focus on common statistical methods used to evaluate machine learning models on binary classification problems.

There are several statistical methods that can be used to evaluate machine learning models. The most common ones are accuracy, precision, recall, and F1-score. These metrics are often used to evaluate classification models, and they are defined as follows:

- **Accuracy**: The proportion of correct predictions out of all predictions made.

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Predictions}}$$

- **Precision**: The proportion of true positive predictions out of all positive predictions made.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

- **Recall**: The proportion of true positive predictions out of all actual positive instances.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

- **F1-score**: The harmonic mean of precision and recall, which balances the two metrics.

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

While accuracy is a simple and intuitive metric, precision, recall and F1-score requires a definition of a positive and negative class. When classifying a binary file as malware or benign, diagnosing disease, or detecting fraud, the positive class is the one we are interested in detecting, and the negative is the one we want to avoid misclassifying. However, in some scenarios, there are no clear positive and negative classes, such as when detecting ISA features in binary files. In these cases, precision, recall, and F1-score can be less meaningful, and accuracy is often the preferred metric.

Accuracy still does not account for the stochastic nature of machine learning, and it is important to use statistical methods to ensure that the measured accuracy is not just due to chance. In binary classification problems without a clear positive class, evaluating estimating confidence intervals for accuracy and running paired t-tests gives a better understanding of the model performance.

2.3.6.1 Confidence intervals for binary classification

Confidence intervals provide a range of values within which the true value is likely to be found, with a specified probability. For example, a 95% confidence interval for the average accuracy across multiple runs with different random initializations means that if we were to repeat the entire experiment many times and compute a confidence interval each time, approximately 95% of those intervals would contain the true average accuracy.

In binary classification tasks, accuracy can be modeled as a binomial process, where each prediction is an independent Bernoulli trial with a probability of successful classification p. The confidence interval is impacted by sources of variability, and when evaluating model performance across multiple seeds, we encounter two sources of uncertainty that needs to be modeled: within-run variance and between-run variance.

2.3.6.1.1 Within-run variance For a single run of a binary classification model with n samples, the accuracy follows a binomial distribution. The variance of the accuracy can be estimated as:

$$\text{Var}_{\text{within\_run}} = \frac{p(1-p)}{n}$$

where p is the observed accuracy and n is the number of test samples.

2.3.6.1.2 Between-run variance Due to the random initialization of models between runs and stochastic training processes, different runs with different seeds will yield different accuracies

on the same dataset. This introduces additional variance where we need to take all accuracies from multiple runs into account. The variance of the accuracy across k runs can be estimated as:

$$\text{Var}_{\text{models}} = \frac{1}{k-1} \sum_{i=1}^{k} (p_i - \bar{p})^2$$

where p_i is the accuracy of run i, $\bar{p}$ is the average accuracy across all runs, and k is the number of runs. We use $k-1$ in the denominator instead of k to account for the degrees of freedom in the sample (Bessel's correction).

2.3.6.1.3 Combined variance and confidence interval construction The total variance combines both sources of uncertainty: the within-run variance and the between-run variance. We are interested in the variance of the average accuracy across k runs. Since we are averaging k independent runs, and the variance of an average of independent random variables decreases by a factor of k, the total variance can be expressed as:

$$\text{Var}_{\text{total}} = \frac{E[\text{Var}_{\text{within\_run}}]}{k} + \frac{\text{Var}_{\text{models}}}{k}$$

This formulation assumes that the within-run and between-run sources of variance are independent, which is reasonable since different random seeds ensure that the stochastic training processes across runs are uncorrelated. The expected value of the within-run variance based on previous assumptions is just the average within-run variance across all runs.

With this combined variance, we can construct a confidence interval for the average accuracy for a given confidence level $(1-\alpha)$. While the central limit theorem states that the distribution of the sample mean approaches a normal distribution as the sample size increases, a more conservative approach for a small number of runs (less than 30) is to use the t-distribution. The confidence interval for k runs can then be constructed as:

$$\text{CI} = \bar{p} \pm t_{\alpha/2,k-1} \cdot \sqrt{\text{Var}_{\text{total}}}$$

where $t_{\alpha/2,k-1}$ is the critical value from the t-distribution for a given confidence level $(1-\alpha)$ and $(k-1)$ degrees of freedom.

2.3.6.1.4 Extension to cross-validation When using cross-validation, the accuracy is calculated for each fold, and the confidence interval can be constructed similarly as in a normal train-test suite. While K-fold cross-validation suites typically have balanced groups of samples, in LOGO CV scenarios the number of test samples within each fold might vary from fold to fold. In this case, the accuracy for each run is calculated as a weighted average of the accuracies across all folds:

$$p_{\text{run}} = \frac{\sum_{j=1}^{f} n_j \cdot p_j}{\sum_{j=1}^{f} n_j}$$

where p_j is the accuracy for fold j, n_j is the number of samples in fold j, and f is the total number of folds. The variance within runs for use in confidence interval construction is then similarly weighted:

$$\text{Var}_{\text{within\_run}} = \frac{\sum_{j=1}^{f} n_j \cdot \frac{p_j(1-p_j)}{n_j}}{\sum_{j=1}^{f} n_j} = \frac{\sum_{j=1}^{f} p_j(1-p_j)}{\sum_{j=1}^{f} n_j}$$

2.3.6.2 Paired t-test for comparing models

A paired t-test provides a statistical approach to determine whether observed differences in performance are statistically significant or could have occurred by chance. When comparing the performance of two models, simply looking at their mean accuracies can be misleading due to the variance inherent in machine learning evaluation. Non-overlapping confidence intervals between two models do give a good indication that the models are statistically different, but overlapping confidence intervals require further analysis to determine whether the difference is statistically significant for a given significance level. The paired t-test is a statistical test that is well suited for model comparison because it can account for correlations between the two models' performance on the same dataset/data splits. For this test, we have the following null and alternative hypotheses:

H_0: There is no difference in mean performance between the two models ($\mu_{\text{diff}} = 0$)
H_1: There is a difference in mean performance between the two models ($\mu_{\text{diff}} \neq 0$)

2.3.6.2.1 Statistical procedure For k paired runs with the same data splits, we calculate the differences in mean accuracy between the two models we want to compare:

$$d_i = p_{i,B} - p_{i,A}$$

where $p_{i,A}$ is the accuracy of model A on run i and $p_{i,B}$ is the accuracy of model B on run i. For the paired t-test, we assume that the differences in performance $d_i = p_{i,B} - p_{i,A}$ are normally distributed. If the number of paired runs is small, we can use the Shapiro-Wilk test to check whether the differences are normally distributed. If they are not, non-parametric alternatives like the Wilcoxon signed-rank test are more suitable. However, the normality assumption is often justified because the differences are computed from the same data splits, reducing variability. The Central Limit Theorem also supports normality when the number of test samples is sufficiently large.

$$d_i \sim N(\mu_{\text{diff}}, \sigma^2_{\text{diff}})$$

Using the normally distributed d_i's, we calculate the paired t-test statistic t which then follows a t-distribution, and is calculated as:

$$t = \frac{\bar{d}}{s_d/\sqrt{k}}$$

where $\bar{d}$ is the mean of the differences, s_d is the standard deviation of the differences, and k is the number of paired runs. The degrees of freedom for the t-distribution is $k - 1$.

We can then calculate the p-value for the t-statistic, which tells us *the probability of observing a difference as extreme as the one we observed, assuming the null hypothesis is true*. We compare the t-value to the t-distribution and calculate the p-value using:

$$\text{p-value} = 2 \cdot P(T_{k-1} \geq |t|)$$

where T_{k-1} represents the t-distribution with $k-1$ degrees of freedom. The factor of 2 accounts for the two-sidedness of the test, since we are interested in whether the difference is significantly different in either direction. If the p-value is below a predefined significance level α, we reject the null hypothesis and conclude that there is a statistically significant difference between the two models. The significance level is typically set to 0.05, meaning that we are willing to accept a 5% chance of incorrectly rejecting the null hypothesis when it is true.

2.3.6.2.2 Extension to cross-validation When using cross-validation, the paired t-test comparisons need an additional aggregation step. For each fold, just like in Section 2.3.6.1.4, we calculate the accuracy for each model in each run as a weighted average of the accuracies across all folds:

$$p_{i,A} = \frac{\sum_{j=1}^{f} n_j \cdot p_{i,j,A}}{\sum_{j=1}^{f} n_j}, \qquad p_{i,B} = \frac{\sum_{j=1}^{f} n_j \cdot p_{i,j,B}}{\sum_{j=1}^{f} n_j}$$

where $p_{i,j,A}$ and $p_{i,j,B}$ is the accuracy of model A and B respectively on fold j in run i, and n_j is the number of samples in fold j. The differences in mean accuracy between the two models are then calculated as:

$$d_i = p_{i,B} - p_{i,A}$$

and the rest of the statistical procedure follows as in the non-cross-validation case described in Section 2.3.6.2.1.

2.4 Related work

In a specialization project leading up to this thesis [8], we conducted a systematic literature review exploring two directions of existing research:

1. Applying machine learning techniques for ISA detection
2. Applications of CNN for binary machine code

Sections 2.4.1 and 2.4.2 summarize our findings from this systematic literature review. Note that parts of the specialization project are revised and reused in these subsections.

Finally, in Section 2.4.3, we describe a master thesis by Andreassen and Morrison from 2024 researching traditional (non-deep) machine learning for detecting specific architectural features from unknown or undocumented ISAs [45].

2.4.1 Machine learning for ISA detection

The goal of the machine learning for ISA detection part of the review was to document previous work that applies machine learning techniques on binaries where the specific ISA of the binaries

were unknown. The part of the process we targeted was before knowledge of the ISA could be used to disassemble or infer other information about the binary. Our search yielded 74 results, which after applying our exclusion and inclusion criteria were filtered down to 6 relevant papers [36, 37, 46, 47, 48, 49].

The existing research we discovered focused on full ISA detection, which is different from our approach. They attempt to classify the ISA of the binary files from a list of known architectures, while we detect individual ISA features independently of the architecture. The existing research also focuses on traditional machine learning techniques and feature engineering, and we found no prior work using deep learning for this task.

2.4.1.1 Feature engineering and feature extraction

The most commonly seen approach to feature engineering for ISA detection relies on byte-level statistical features, with Byte Frequency Distribution (BFD) being the most notable technique. BFD was first introduced by Clemens and adopted by multiple subsequent studies including ELISA, Beckman & Haile, and ISAdetect [36, 37, 46, 47]. The technique involves counting occurrences of each possible byte value in a 256-dimensional feature vector and normalizing by binary size to account for different program sizes. This approach achieved up to 94.02% accuracy in 10-fold cross-validation, demonstrating that BFD preserves sufficient ISA-specific information for classification. However, BFD struggles when distinguishing architectures that differ only in endianness, such as MIPS and MIPSEL, where F1-scores dropped to ~0.47 compared to over 0.98 for other architectures. To address this, Clemens developed endianness heuristics based on common immediate value patterns (increment/decrement operations encoded as `0x0001`/`0xFFFE` in big-endian versus `0x0100`/`0xFEFF` in little-endian), expanding the feature vector to 260 dimensions and improving overall accuracy from ~93% to ~98% [46].

Other approaches include Sahabandu et al.'s N-gram TF-IDF method inspired by natural language processing, which uses term frequency-inverse document frequency for all 1-grams, 2-grams, and the top 5000 3-grams. This results in a much larger feature vector of 70,792 dimensions compared to Clemens' 260, but achieving accuracies of 99% and 98% on different datasets [49]. Ma et al. took a domain-specific approach for grid device firmware, leveraging the knowledge that most such devices use RISC instruction sets with 4-byte instruction widths, dividing binaries into 4-byte chunks, and applying text classification techniques [48]. While some studies like ELISA and ISAdetect expand on Clemens's work by incorporating architecture-specific features such as function prologues, epilogues, and known signatures from the angr binary analysis platform to improve accuracy, this approach requires manual feature definitions for each target ISA, potentially limiting scalability to new or custom architectures commonly found in IoT and embedded systems [36, 37].

2.4.1.2 Machine learning techniques

The machine learning approaches in the literature for ISA detection rely on traditional, non-deep learning algorithms, with Support Vector Machines (SVM) standing out as the best-performing classifier across studies. SVM was evaluated in five out of six reviewed papers and achieved the highest accuracies like the 98.35% score in Clemens' study, closely followed by Decision Tree and Logistic Regression models [36, 46, 47, 48, 49]. Ma et al. attribute SVM performance to its ability to separate complex non-linear boundaries efficiently while remaining less prone to overfitting [48]. SVM's capacity to achieve high accuracy with relatively small amounts of training data is also

noted by multiple researchers. While ISAdetect found Random Forest slightly more accurate on expanded feature sets [36], and ELISA used only Logistic Regression without detailed justification [37], the overall trend shows linear classifiers like SVM work well for this task. However, the reviewed studies spent little time on model selection or hyperparameter optimization details, instead focusing on feature engineering approaches. This suggests that the effectiveness of these methods largely stems from their ability to capture ISA information from engineered features, rather than from new machine learning inventions.

2.4.1.3 Datasets and classification targets

All six reviewed papers focused on multi-class classification of instruction set architectures from predefined sets of known ISAs, ranging from 3 to 23 different architectures, rather than attempting to detect specific ISA features in isolation. The most commonly targeted architectures were RISC-based ISAs including ARM32, ARM64, MIPS, and PowerPC (represented in all studies), along with the well-known CISC-based architecture x86-64. The widest architecture coverage was found in studies by Clemens (20 ISAs), ELISA (21), ISAdetect (23), and Sahabandu et al. (12-23 across datasets) [36, 37, 46, 49]. In these studies, byte-level N-gram approaches show no significant differences in accuracy when comparing different architecture types, such as 32-bit versus 64-bit word sizes or RISC versus CISC instruction sets. The primary classification difficulties appeared to come from architectures with similar opcodes but different endianness configurations, leading researchers to develop specific endianness detection features mentioned in Section 2.4.1.1. Even though these endianness features were developed, none evaluated endianness detection in isolation or across architectures supporting multiple endianness configurations.

An important limitation identified across the studies relates to their approaches to training and evaluation of code sections versus whole binary files, which has implications for real-world applicability. While some studies like Clemens used ELF headers to identify and train exclusively on code sections [46], others like Ma et al. acknowledged the potential absence of header information in real-world scenarios but failed to specify their testing methodology in regard to this [48]. Interestingly, ISAdetect's more comprehensive evaluation on this revealed a large performance degradation when testing on whole binaries compared to code-only sections, with SVM accuracy dropping from 99.7% to 73.2% [36]. This indicates that data and other non-code sections introduced detrimental noise to byte-level N-gram features. To address this challenge, ELISA developed a two-phase approach using Conditional Random Fields for code section identification, though this required correct ISA classification first [37]. Beckman & Haile introduced an "architectural agreement" method using sliding windows to identify code sections based on consistent architectural predictions [47]. The different approaches of addressing code-only sections versus full binaries highlight the difficulties in automatically analyzing binaries without reliable section information.

2.4.2 CNN applications for binary machine code

Although CNNs are traditionally applied to vision tasks such as image classification or object recognition, prior research has also used CNNs for analyzing binary machine code. Notably, research on malware detection and classification has proven that CNNs can perform well on raw binary code.

2.4.2.1 Malware classification

The Microsoft Malware Classification Challenge (MMCC) dataset was published as part of a research competition in 2015 and contains malware binaries from 9 different malware families [50]. Multiple researchers have used this dataset and developed CNN architectures for distinguishing the different classes of malware. A summary of said literature and their classification performance is shown in Table 2.2.

Table 2.2: Microsoft Malware dataset classification performance.

Paper (year published)	Accuracy	Precision	Recall	F1-score
Rahul et al. [51] (2017)	0.9491	-	-	-
Kumari et al. [52] (2017)	0.9707	-	-	-
Yang et al. [53] (2018)	0.987	-	-	-
Khan et al. [54] (2020)	0.9780	0.98	0.97	0.97
Sartoli et al. [55] (2020)	0.9680	0.9624	0.9616	0.9618
Bouchaib & Bouhorma [56] (2021)	0.98	0.98	0.98	0.98
Liang et al. [57] (2021)	0.9592	-	-	-
SREMIC [58] (2024)	**0.9972**	**0.9993**	**0.9971**	**0.9988**

Malimg is another dataset containing malware from 25 different families [59]. As opposed to MMCC, this dataset contains binaries pre-encoded to an image format, using each byte in the file as a single pixel value. A summary of identified research applying CNNs to this dataset is shown in Table 2.3.

Table 2.3: Malimg dataset classification performance.

Paper (year published)	Accuracy	Precision	Recall	F1-score
Cervantes et al. [60] (2019)	0.9815	-	-	-
El-Shafai et al. [61] (2021)	**0.9997**	0.9904	0.9901	0.9902
Li et al. [62] (2021)	0.97	-	-	-
Son et al. [63] (2022)	0.97	-	-	-
Hammad et al. [64] (2022)	0.9684	-	-	-
S-DCNN [65] (2022)	0.9943	0.9944	0.9943	0.9943
SREMIC [58] (2024)	0.9993	**0.9992**	**0.9987**	**0.9987**
DCMN [66] (2024)	0.9989	0.9971	0.9984	0.9977

The existing studies differ in their approach to data encoding and model architecture. Certain studies used a one-dimensional vector encoding of the binary data, where each byte in the binary file is converted to a decimal value between 0 and 255. The encoded bytes were then passed to a one-dimensional CNN for classification [51, 62]. However, the most common approach was to encode the binary data as a two-dimensional image, where each byte in the binary file is converted to a pixel value. These grayscale images are then used for input to traditional two-dimensional CNN architectures [52, 53, 56, 57, 61, 63, 64, 66, 67].

2.4.2.2 Compiler optimization detection

Compilers such as GCC allow the user to choose between five general optimization levels: -O0, -O1, -O2, -O3, and -Os. Knowing which of these levels was used for compilation can be useful in areas such as vulnerability discovery. We identified two prior studies that attempt to detect a binary's compiler optimization level.

Yang et al. achieved an overall accuracy of 97.24% on their custom dataset, with precision for each class ranging from 96% to 98% [68]. This was a significant improvement compared to previous research on detecting compiler optimization levels. Pizzolotto & Inoue elaborated on this work by using binaries compiled across 7 different CPU architectures, as well as compiling with both GCC and Clang for the x86-64 and AArch64 architectures [69]. They showed a 99.95% accuracy in distinguishing between GCC and Clang, while the optimization level accuracy varies from 92% to 98% depending on the CPU architecture. However, we note that Pizzolotto & Inoue treated -O2 and -O3 as separate classes, whereas Yang et al. considered these as the same class, making the comparison slightly unfair.

Both studies used a one-dimensional vector encoding of the binary data, where each byte in the binary file is converted to a decimal value between 0 and 255. The encoded bytes were then passed to a one-dimensional CNN for classification.

2.4.3 Classifying ISA features with machine learning

Joachim Andreassen's master's thesis from 2024 explored the detection of architectural features from binary programs where the ISA is unknown or undocumented [45]. Unlike the studies discussed in Section 2.4.1, which applied machine learning techniques to ISA detection, Andreassen attempted to detect endianness and instruction width from binaries of previously unseen ISAs.

The methods applied in this thesis are traditional, non-deep machine learning models such as random forests, k-nearest neighbors, and logistic regression. It makes heavy use of feature engineering, using existing features from prior literature and engineering novel ones, particularly for detecting instruction width. Some of these custom features use signal processing techniques such as autocorrelation and Fourier transformation.

Using Leave-One-Group-Out Cross-Validation (LOGO CV) on the ISAdetect dataset, training on only the code section part of the binary file, the best-performing models achieve an average accuracy of 92.0% on endianness detection and 86.0% in distinguishing between fixed and variable instruction width. We will use Andreassen's thesis as a comparison for determining whether our CNN approach can eliminate the need for manual feature engineering, without sacrificing model performance.

Chapter 3

Methodology

This chapter describes the methodology used in this thesis. We start by describing the experimental setup, including the system configuration and the datasets used in the thesis in Section 3.1. We then outline the development process for BuildCross, our custom dataset, in Section 3.2. Next, we describe the machine learning models, target features, and data preprocessing used in the experiments in Section 3.3. Finally, we present our evaluation strategy and metrics in Section 3.4.

3.1 Experimental setup

3.1.1 Datasets

This thesis utilizes three primary datasets: BuildCross, ISAdetect, and CpuRec. The BuildCross dataset is a novel contribution of this thesis, and its development is discussed in Section 3.2. ISAdetect and CpuRec datasets are sourced from previous work in software reverse engineering. These datasets contain samples of binary programs from a variety of different Instruction Set Architectures (ISAs). Architectures vary in their similarity regarding features such as endianness, word size, and instruction width, and our model development focuses on the ability to reliably detect architectural features independent of the specific ISA. The choice of datasets is therefore motivated by architectural diversity, with the goal of reducing potential correlations between groups of ISAs and the features we aim to detect. Additionally, since binary programs are not human-readable, errors and inconsistencies in the data are difficult to uncover. We depend on accurate labeling of the datasets to ensure reliable results. Based on our search for appropriate datasets, we have found that the combination of the ISAdetect and CpuRec datasets strikes a balance between the number of architectures represented and the volume of training data available, and they complement each other in a way that aligns with our research objectives.

3.1.1.1 ISAdetect

The ISAdetect dataset is the product of a master's thesis by Sami Kairajärvi and the resulting paper *ISAdetect: Usable Automated Detection of CPU Architecture and Endianness for Executable Binary Files and Object Code* [36]. One of their key contributions is providing, to our knowledge, the most comprehensive publicly available dataset of binary programs from different ISAs to date. All program binaries were sourced from Debian Linux repositories, chosen because Debian is a trusted project that has been ported to a wide range of ISAs. This resulted in a dataset consisting of 23 dif-

ferent architectures. Kairajärvi et al. also focused on addressing the dataset imbalances found in previous research, such as Clemens' work, with each architecture containing approximately 3,000 binary program samples [46]. Table 3.1 lists the ISAs present in ISAdetect and their architectural features.

Table 3.1: ISAs present in ISAdetect dataset

ISA	Endianness	Word size	Instruction width
alpha	little	64	32
amd64	little	64	variable
arm64	little	64	32
armel	little	32	32
armhf	little	32	32
hppa	big	32	32
i386	little	32	variable
ia64	little	64	variable
m68k	big	32	variable
mips	big	32	32
mips64el	little	64	32
mipsel	little	32	32
powerpc	big	32	32
powerpcspe	big	32	32
ppc64	big	64	32
ppc64el	little	64	32
riscv64	little	64	32
s390	big	32	variable
s390x	big	64	variable
sh4	little	32	16
sparc	big	32	32
sparc64	big	64	32
x32	little	32	variable

The ISAdetect dataset is publicly available through etsin.fairdata.fi [70]. Our study utilizes the most recent version available at the time of running our experiments, Version 6, released on March 29, 2020. The dataset is distributed as a compressed archive (new_new_dataset/ISAdetect_full_dataset.tar.gz) containing both complete program binaries and code-only sections for each architecture. Additionally, each ISA folder contains a JSON file with detailed metadata for each binary, including properties such as endianness and word size. Additional labeling used by this thesis was based on another master's thesis by Andreassen and Morrison [45]. We also expanded the instruction width labeling for undocumented architectures by looking up technical documentation and manuals for the ISAs in question. Our full set of labels and the differences from prior work are documented in Table A.4 in the appendix.

3.1.1.2 CpuRec

The CpuRec dataset is a collection of code-only sections extracted from binaries of 72 different ISAs. It was developed by Louis Granboulan for use with the *cpu_rec* tool, which employs Markov

chains and Kullback-Leibler divergence to classify the ISA of input binaries [71]. Although only one binary per architecture is provided – which is likely insufficient for training a deep learning model on its own – the diversity of ISAs represented makes this an excellent testing dataset for evaluating our models. Table 3.2 lists the ISAs present in CpuRec and their architectural features.

Table 3.2: ISAs present in CpuRec dataset

ISA	Endianness	Word size	Instruction width
X86	little	32	variable
X86-64	little	64	variable
ARM64	little	64	32
Alpha	little	64	32
ARMel	little	32	32
ARMhf	little	32	32
MIPSeb	big	32	32
MIPSel	little	32	32
PPCeb	big	32	32
PPCel	little	64	32
HP-PA	big	32	32
IA-64	little	64	variable
M68k	big	32	variable
RISC-V	little	64	32
S-390	big	64	variable
SuperH	little	32	16
SPARC	big	unknown	32
ARC32el	little	32	variable
AxisCris	little	32	16
Epiphany	little	32	variable
M88k	big	32	32
MMIX	big	64	32
PDP-11	middle	16	variable
Stormy16	bi	32	variable
V850	little	32	variable
Xtensa	bi	32	variable
6502	little	8	variable
ARcompact	little	32	variable
Blackfin	little	32	variable
FR30	big	32	16
i860	bi	32	32
MCore	big	32	16
MN10300	little	32	unknown
PIC10	n/a	8	12
RL78	little	unknown	unknown
VAX	little	32	variable
XtensaEB	big	32	variable
68HC08	big	8	variable
Cell-SPU	bi	64	unknown

ISA	Endianness	Word size	Instruction width
FR-V	unknown	32	32
Mico32	big	32	32
Moxie	bi	32	variable
PIC16	n/a	8	14
ROMP	big	32	variable
TILEPro	unknown	32	variable
Visium	unknown	unknown	unknown
Z80	little	8	variable
68HC11	big	8	variable
ARMeb	big	32	32
CLIPPER	little	32	variable
FT32	unknown	32	unknown
IQ2000	big	32	unknown
MicroBlaze	big	32	32
MSP430	little	16	variable
PIC18	n/a	8	16
RX	little	32	variable
TLCS-90	n/a	8	variable
8051	n/a	8	variable
CompactRISC	little	16	variable
H8-300	n/a	8	variable
M32C	little	32	variable
MIPS16	bi	16	16
NDS32	little	32	variable
PIC24	little	16	24
TMS320C2x	unknown	16/32	variable
WE32000	n/a	32	unknown
Cray	n/a	64	variable
H8S	unknown	16	variable
M32R	bi	32	variable
NIOS-II	little	32	32
TMS320C6x	bi	32	32
ARC32eb	little	32	variable
AVR	n/a	8	variable
HP-Focus	n/a	32	variable
STM8	n/a	8	variable
TriMedia	unknown	32	unknown

The cpu_rec tool suite is available on GitHub, and the binaries used in this thesis are available in the cpu_rec_corpus directory within the cpu_rec repository [72]. The dataset was curated from multiple sources. A significant portion of the binaries were sourced from Debian distributions, where more common architectures like `x86`, `x86_64`, `m68k`, `PowerPC`, and `SPARC` are available. For less common architectures, binaries were collected from the Columbia University Kermit archive, which provided samples for architectures such as `M88k`, `HP-Focus`, `Cray`, `VAX`, and `PDP-11`. The remaining samples were obtained through compilation of open-source projects using GNU Com-

piler Collection (GCC) cross-compilers [71]. Unlike ISAdetect, the CpuRec dataset only labels the name of the ISA, without additional architectural features.

However, the documentation of the CpuRec dataset is not as comprehensive as ISAdetect, and the authors of the CpuRec dataset have not provided detailed information about how the binaries were sourced. To address this gap, we referenced Appendix A in the thesis by Andreassen and Morrison, who used this dataset in their work and provided labels for architectural features including endianness, word size, and instruction width specifications for each architecture in the dataset [45]. While we relied on the labeling work by Andreassen and Morrison, we also reviewed technical documentation and manuals available online for all the architectures in question to verify our labeling. Sources used and conclusions drawn in this process are documented in the CSV file used in our source code (`/masterproject/code/dataset/cpu_rec-features.csv`) [73]. We provide a more detailed discussion on dataset quality in Section 5.5.2. Our labels differ from those of Andreassen and Morrison, and a comparison between them can be found in Table A.5 in the appendix.

3.1.2 Technical configuration

For all experiments, we use the Idun cluster at the Norwegian University of Science and Technology (NTNU). This High Performance Computing (HPC) cluster is equipped with 230 NVIDIA Data Center Graphics Processing Units (GPUs) [74]. The following hardware configuration was used for all experiments:

- CPU: Intel Xeon or AMD EPYC (12 cores enabled)
- GPU: NVIDIA A100 40GB
- RAM: 16 GB

We use the PyTorch framework for building and training our models. The following software versions were used:

- Python 3.12.3
- PyTorch 2.2.2
- torchvision 0.17.2
- CUDA 12.1
- cuDNN 8.9.2

3.1.3 Hyperparameters

Unless specified otherwise, we use the training hyperparameters specified in Table 3.3 for our experiments.

Table 3.3: Hyperparameter selection

Hyperparameter	Value
Batch size	64
Loss function	Cross entropy
Optimizer	AdamW
Learning rate	0.0001
Weight decay	0.01
Number of epochs	2

We find that a batch size of 64 represents a good balance between computational efficiency and model performance. It is large enough to enable efficient GPU utilization, while small enough to provide a regularization effect through noise in gradient estimation.

Cross entropy loss is the natural choice for classification tasks, as it tends to provide superior performance for classification tasks compared to mean squared error loss [75].

The AdamW optimizer is an improved version of Adam that implements weight decay correctly, decoupling it from the learning rate. It also improves on Adam's generalization performance on image classification datasets [76].

A learning rate of 0.0001 is lower than Pytorch's default of 0.001 for AdamW. We make this conservative choice due to early observations showing that small learning rates still cause the AdamW optimizer to reach convergence rather quickly for our dataset. Considering our vast amounts of computational resources, we want to err on the side of slower training rather than risking convergence issues.

A weight decay of 0.01 provides moderate regularization strength and offers a balance between underfitting and overfitting. It is Pytorch's default for the AdamW optimizer.

We find that Convolutional Neural Network (CNN) models trained on our datasets converge after just 2 epochs, and further training does not decrease the training loss nor improve the validation accuracy.

3.2 Developing a custom dataset

This thesis introduces BuildCross, a comprehensive toolset and diverse program binary dataset for use by machine learning models in binary analysis. It compiles and extracts code sections from archive files of widely-used open-source libraries (referenced in Table 3.4). The code sections in the binary files are extracted, in addition to being disassembled for dataset labeling and quality control. We develop BuildCross to bridge the gap in ISA diversity and volume between the ISAdetect and CpuRec datasets. While ISAdetect contains a large volume of binary programs, it consists mostly of architectures from mainstream ISAs. We believe this dataset alone lacks sufficient diversity to develop truly architecture-agnostic models. CpuRec, on the other hand, contains binaries from a great variety of architectures, but the lack of significant volume and uncertainties with the labeling of the dataset makes it unsuited for training larger machine learning models. BuildCross strikes a balance between the two, aiming to generate a larger volume of binary code for the underrepresented, less common architectures.

We have found that large, consistent sources of precompiled binaries for embedded and bare-metal systems are hard to come by, a notion also shared by the authors of ISAdetect and CpuRec [36, 71]. To overcome this limitation and produce a well-documented, correctly labeled dataset, we compile binary programs for uncommon architectures using cross-compilation with GCC and GNU Binutils. We develop a pipeline consisting of three steps:

1. Creating containerized cross-compiler toolchains for different ISAs.
2. Gathering compilable source code, configuring the toolchains, and compiling binaries.
3. Extracting features and relevant data from the compiled libraries.

Given ISAdetect's comprehensive architectural coverage, we focus on ISAs not included in their dataset. Our pipeline is extendable and can incorporate additional target toolchains and binary sources as needed.

3.2.1 Pipeline for developing toolchains

In order to generate binary programs for specific ISAs, we need a cross-compiler that can run on our host system and target that architecture. While common targets like x86, ARM and MIPS systems have readily available toolchains for multiple host platforms, the less common architectures not covered by the ISAdetect dataset are, in our experience, either not publicly available or cumbersome to configure properly. We found that the best option was to create and compile these toolchains ourselves, and we decide on GCC and GNU Binutils due to the GNU project's long history of supporting a large variety of architectures.

A full cross-compiler toolchain has numerous parts, and since many architectures are unsupported in newer versions of GCC, configuring compatible versions of Binutils, LIBC implementations, GMP, MPC, MPFR, etc. would require much trial and error. To get started, we employ the buildcross project by Mikael Pettersson on GitHub, as it contains a setup for building cross-compilers with documented version compatibility for deprecated architectures [77]. The buildcross project is used as a base for our toolchain building scripts, and is expanded to support additional architectures.

The cross-compiler suite uses Apptainer images to create containerized, reproducible, and portable cross-compilation environments for the supported architectures. Apptainer is an open-source alternative to Docker and is designed for high-performance computing environments [78, 79]. The GCC suite's source code with its extensions is around 15GB in size, and to reduce image space and build time, we create a builder image with the necessary dependencies and libraries for building the toolchains. This builder script is used to build the toolchain for each architecture, and the resulting toolchains are stored in separate images of roughly 500MB in size.

3.2.2 Configuring toolchains and gathering library sources

To streamline the compilation process across multiple target architectures, we use CMake toolchain configuration files rather than manually configuring each library. The manual approach of configuring each library individually for every target architecture is both time-consuming and prone to inconsistencies. CMake, a widely used build system, simplifies this process by allowing us to configure and generate build files for different platforms and compilers in a platform-independent way [80]. With CMake, we can specify the target architecture, compiler, linker, and other build options using just one toolchain file per architecture. While most architectures can share a common template toolchain file, CMake also makes it straightforward to implement specific configurations for architectures with unique requirements.

The libraries we select for our dataset are widely used and have large codebases, which provide a good representation of real-world code. They are chosen to ensure that the generated binaries are representative of actual software applications. This is important for training and evaluating our models, as it allows us to assess their performance on realistic data. Additionally, using well-known open-source libraries helps us avoid potential issues with licensing, distribution, and reproducibility. By compiling these libraries for the target architectures, we can create a diverse dataset that covers a wide range of instruction sets and architectural features. With the only re-

quirement being that the libraries support CMake, the BuildCross suite can also accommodate additional libraries in the future.

Table 3.4: Source libraries used to compile and generate the BuildCross dataset.

Library	Version	Description
freetype [81]	2.13.3	A software library for rendering fonts. It is widely used for high-quality text rendering in applications, providing support for TrueType, OpenType, and other font formats [82].
libgit2 [83]	1.9.0	A portable, pure C implementation of the Git core methods. It provides a fast, linkable library for Git operations that can be used in applications to implement Git functionality without spawning a git process [84].
libjpeg-turbo [85]	3.1.0	An optimized version of libjpeg that uses SIMD instructions to accelerate JPEG compression and decompression. It is significantly faster than the original libjpeg while maintaining compatibility [86].
libpng [87]	1.6.47	The official PNG reference library that provides support for reading, writing, and manipulating PNG (Portable Network Graphics) image files. It is widely used in graphics processing applications [88].
libwebp [89]	1.5.0	A library for encoding and decoding WebP images, Google's image format that provides superior lossless and lossy compression for web images, resulting in smaller file sizes than PNG or JPEG [90].
libyaml [91]	0.2.5	A C library for parsing and emitting YAML data. It is commonly used in configuration files and data serialization applications [92].
pcre2 [93]	10.45	Perl Compatible Regular Expressions library (version 2), which provides functions for pattern matching using regular expressions. It is used in many applications for text processing and search operations [94].
xzutils [95]	5.7.1	A set of compression utilities based on the LZMA algorithm. The XZ format provides high compression ratios and is commonly used for software distribution and archiving [96].
zlib [97]	1.3	A software library used for data compression. It provides lossless data-compression functions and is widely used in many software applications for compressing data, including PNG image processing [98].

The toolchain configuration setup has some flaws, as some of the libraries have dependencies that are not compatible with the target architecture. This is especially true for libraries that are not actively maintained, and the manual labor of patching libraries for each architecture does not scale well for the high number of ISAs. The most common issue we encounter is the lack of libc intrinsic header file definitions for some of the targets. CMake can in some cases be used to disable some of the library features with missing dependencies, at the cost of in some cases reducing code size. We also compile most architectures with the linker flag `-Wl,--unresolved-symbols=ignore-all`,

creating binaries that most likely would crash at runtime if the missing symbols were used. Ignoring missing symbols and similar shortcuts still produces valid binaries that are useful for our dataset, as the goal is to create a dataset that is representative of the architectures and their features. Despite this, not all libraries can be compiled for all architectures in time for this thesis, which explains the discrepancies in the amount of data between the architectures.

3.2.3 Gathering results

The final stage of our pipeline involves extracting and labeling binary data from the compiled libraries. Using CMake's configuring, building, and installing features, we generate install folders containing compiled archive files (.a) for each target architecture. These archive files are collections of compiled binaries (object files) in Executable and Linkable Format (ELF) format, providing functions and utilities other programs can link to.

Using the GNU Binutils toolkit from our compiled toolchains, we employ the *archiver* (ar) to extract individual object files, *objcopy* to isolate code sections from these objects, and *objdump* to generate disassembly. This process yields our core dataset of compiled code sections across all target architectures.

For dataset labeling, we extract the endianness and word size metadata directly from each architecture's ELF headers. However, determining instruction width proved more challenging due to inconsistent online documentation for uncommon architectures. We establish a method of analyzing instruction patterns in the disassembly, using the hexadecimal mapping between instructions and assembly to infer the size of the instructions. The disassembly output is included in the dataset both for verification of our labeling and as an added utility for the use of BuildCross.

3.2.4 Final dataset yields and structure

The final dataset spans 40 architectures with approximately 120 MB of binary code. Information on the included architectures can be found in Table 3.5. The amount of data varies across the architectures, with more supported architectures like arc, loongarch64, and blackfin containing more files, while rarer architectures like xstormy16 and rl78 contain fewer samples due to compilation challenges mentioned in Section 3.2.2.

The source code for the cross-compiler suite is available under the masterproject/crosscompiler directory on the thesis GitHub page [73]. The dataset itself is published as a GitHub Release and distributed as a tar.gz file with the following structure:

```
buildcross_dataset.tar.gz.
├── library_files/ (Full compiled libraries)
│   ├── arc/
│   ├── bfin/
│   └── (…)
├── text_asm/ (Decompiled code sections of libraries)
│   ├── arc/
│   ├── bfin/
│   └── (…)
├── text_bin/ (Raw binary of code sections)
│   ├── arc/
│   ├── bfin/
│   └── (…)
├── labels.csv (Dataset labels for endianness, word size, and instruction width)
└── report.csv (Code section file sizes for each library in csv format)
```

└── report.txt (Code section file sizes for each library in text format)

The `labels.csv` file contains architecture metadata including endianness, word size and instruction width for each binary. The report files provide detailed statistics on code section sizes across libraries, with `report.csv` offering machine readable format and `report.txt` providing human-readable summaries. The data from the `labels.csv` file is presented in Table 3.5.

Table 3.5: Labels for the ISAs in the BuildCross dataset, with documented feature values for endianness, word size, instruction width type, and instruction width. Also includes code section sizes extracted for each architecture

architecture	endianness	word size	instruction width type	instruction width	total size (kb)
arc	little	32	variable	16/32	3299
arceb	big	32	variable	16/32	1729
bfin	little	32	variable	16/32	2942
c6x	big	32	fixed	32	5271
cr16	little	32	variable	16/32	1583
cris	little	32	variable	16/32	4070
csky	little	32	variable	16/32	4244
epiphany	little	32	variable	16/32	334
fr30	big	32	variable	16/32	2215
frv	big	32	fixed	32	5033
ft32	little	32	fixed	32	445
h8300	big	32	variable	16/32	4396
iq2000	big	32	fixed	32	2459
kvx	little	64	variable	16/32	5012
lm32	big	32	fixed	32	3392
loongarch64	little	64	fixed	64	4814
m32r	big	32	fixed	32	1997
m68k-elf	big	32	variable	16/32/48	1866
mcore	little	32	fixed	16	1268
mcoreeb	big	32	fixed	16	1268
microblaze	big	32	fixed	64	5862
microblazeel	little	32	fixed	64	5834
mmix	big	64	fixed	32	4305
mn10300	little	32	variable	na	1251
moxie	big	32	variable	16/32	2236
moxieel	little	32	variable	16/32	2229
msp430	little	32	variable	16/32	223
nds32	little	32	variable	16/32	2507
nds32be	big	32	variable	16/32	1431
nios2	little	32	fixed	64	4299
or1k	big	32	fixed	64	5541
pru	little	32	fixed	32	2435
rl78	little	32	variable	na	338
rx	little	32	variable	na	1486
rxeb	big	32	variable	na	1300

architecture	endianness	word size	instruction width type	instruction width	total size (kb)
tilegx	little	64	fixed	64	11964
tilegxbe	big	64	fixed	64	11970
tricore	little	32	variable	16/32	1644
v850	little	32	variable	16/32	3171
visium	big	32	fixed	32	3481
xstormy16	little	32	variable	16/32	219
xtensa	big	32	variable	na	2671

3.3 Experiments

This research primarily involves training, validating, and evaluating CNN models using the ISA characteristics endianness and instruction width as the target features. This section outlines our approach to data preprocessing as well as the model architectures we use for our experiments.

3.3.1 Data preprocessing

While most CNN architectures are designed for image data, our datasets consist of compiled binary executables. Thus, how these are encoded into a format that can be consumed by a CNN is a crucial part of our methodology. In our experiments, we use two different approaches for image encoding.

3.3.1.1 Two-dimensional byte-level encoding

Using two-dimensional byte-level encoding, we treat each byte in the binary file as an integer with values ranging from 0 to 255. These values are arranged in a two-dimensional array of predetermined size and fed into the CNN. For files larger than this predetermined size, we use only the initial bytes that fit within the dimensions. Rather than applying data augmentation techniques for files smaller than this predetermined size, we exclude them from the dataset entirely. We discuss this approach in more detail in Section 3.3.1.3.

When applying two-dimensional CNN on 2D grids of this format, the byte values will essentially be treated as pixel values, where the byte sequence forms a grayscale image. Figure 3.1 shows an example of a 9-byte sequence encoded as a 3x3 pixel grayscale image.

This approach was chosen based on previous literature which successfully classified malware from binary executables using CNNs [52, 56, 57, 61, 63, 64, 66, 67].

3.3.1.2 One-dimensional byte-level encoding

Similar to the 2D approach, in one-dimensional byte-level encoding we treat each byte as an integer value and place them in a one-dimensional array of a predetermined size. If the file is larger than the predetermined size, only the first bytes are used. If the file is smaller than the predetermined size, they are excluded from the dataset.

This approach was chosen based on previous literature which successfully detected compiler optimization levels in binary executables using 1D CNNs [68, 69].

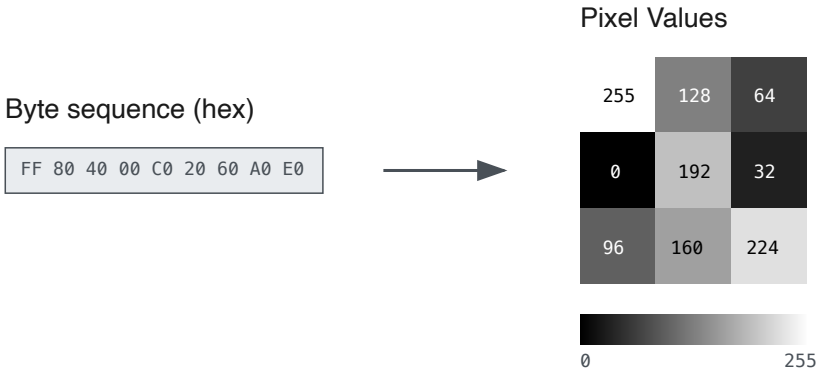


Figure 3.1: Encoding bytes as a grayscale image.

3.3.1.3 Input size and file splitting

Our datasets contain binary files of vastly different sizes, requiring methods to handle this variability. When training our model, we want each data instance to be of a fixed size. Our goal is to use input sizes as small as possible to create time and energy-efficient models, while still being large enough to capture enough information about the features we aim to detect. In addition, smaller input sizes ensure that most of our files are large enough to fill the entire input vector, increasing the amount of usable data in the dataset.

For the one-dimensional models, we pick an input size of 512 bytes. We choose this number based on preliminary testing, which revealed that input sizes larger than 512 bytes did not improve model performance.

For the small two-dimensional models, we use an input size of 32x16. This matches the 512-byte input size for the one-dimensional models. In addition, we hypothesize that using a width of 32 bytes might improve the models' ability to detect repeating patterns, since many programs use an instruction width of 32. For the models based on ResNet, we use an input size of 32x32 (which gives 1024 bytes), since this architecture is designed for square images.

For files smaller than the predetermined input size, we choose to exclude them from training rather than padding them to fit the input size. Only the ISAdetect dataset contains files below this threshold, and these represented a small, likely insignificant portion of the total files in that dataset. Furthermore, data augmentation techniques such as padding can introduce noise, and we concluded that exclusion was a better option than potentially degrading the quality of the data. For files in ISAdetect and CpuRec datasets larger than the input size I, only the first I bytes from the file are used. For the custom dataset we developed, which contains few but rather large files, we use file splitting to increase the number of training instances. Since these binary files are composed of concatenated code sections from multiple library object files, we consider file splitting to be an appropriate approach. Given a file of size F and a model input size of I, each file is divided into $\lfloor F/I \rfloor$ instances.

3.3.2 Model architectures

In our experiments, we train, evaluate, and compare several model architectures to detect architectural features from binary code. Our approach is inspired by successful applications of CNNs

to binary analysis in previous work, detailed in Section 2.4.2. While the scope of this thesis limits the range of model variations we can test, we focus on three factors that have influenced model performance in previous research: input encoding, model size and complexity, and embedding layers.

The model architectures described below are specifically designed to investigate how these factors affect a model's ability to learn target ISA features in relation to our RQs. By systematically varying these aspects across our experiments, we aim to determine how effective they are in determining ISA features from binary programs.

3.3.2.1 Simple 1D CNN

The smallest model we developed is a small one-dimensional CNN. The first layer is a 1x1 convolution layer, bringing the filter space dimensionality from 1 to 128 while keeping the spatial dimensions. The rationale for this layer is to align the feature space with the embedding model introduced in Section 3.3.2.2. Then, the model consists of three convolutional blocks, each with two convolutional layers and a max pooling layer. After the convolutional blocks comes a global average pooling layer, and a fully-connected block with a single hidden layer for classification. Dropout with a rate of 0.3 is applied after each convolution block and between the two fully-connected layers. The full model specification is shown in Table 3.6. The model has a total of 152,282 trainable parameters and is hereby referred to as *Simple1d*.

Table 3.6: Simple 1D CNN

Layer	Hyperparameters	Output Shape	Parameters
Input	–	(512,)	–
Convolution 1D	k=1, s=1, p=0	(512, 128)	256
Dropout	p=0.3	(512, 128)	–
Convolution 1D	k=3, s=1, p=1	(512, 32)	12,320
Convolution 1D	k=5, s=2, p=2	(256, 32)	5,152
Max Pooling 1D	k=2, s=2	(128, 32)	–
Dropout	p=0.3	(8, 128)	–
Convolution 1D	k=3, s=1, p=1	(128, 64)	6,208
Convolution 1D	k=5, s=2, p=2	(64, 64)	20,544
Max Pooling 1D	k=2, s=2	(32, 64)	–
Dropout	p=0.3	(8, 128)	–
Convolution 1D	k=3, s=1, p=1	(32, 128)	24,704
Convolution 1D	k=5, s=2, p=2	(16, 128)	82,048
Max Pooling 1D	k=2, s=2	(8, 128)	–
Dropout	p=0.3	(8, 128)	–
Adaptive Avg Pool 1D	output=1	(1, 128)	–
Reshape	–	(128,)	–

Layer	Hyperparameters	Output Shape	Parameters
Fully Connected	–	(8,)	1,032
ReLU	–	(8,)	–
Dropout	p=0.3	(8,)	–
Fully Connected	–	(2,)	18
Softmax	–	(2,)	–

3.3.2.2 Simple 1D CNN with embedding layer

Our one-dimensional word-embedding model builds on the *Simple1d* CNN in Section 3.3.2.1, and is constructed by placing an embedding layer at the beginning of the model instead of the 1x1 convolution layer. The embedding layer transforms the byte values into a vector of continuous numbers, allowing the model to learn the characteristics of each byte value and represent it mathematically. After the embedding layer, the model is identical to the *Simple1d* model. The full model specification is shown in Table 3.7. This model has a total of 184,794 trainable parameters and is hereby referred to as *Simple1d-E*.

Table 3.7: Simple 1D CNN with embedding layer

Layer	Hyperparameters	Output Shape	Parameters
Input	–	(512,)	–
Embedding	v=256, d=128	(512, 128)	32,768
Dropout	p=0.3	(512, 128)	–
Convolution 1D	k=3, s=1, p=1	(512, 32)	12,320
Convolution 1D	k=5, s=2, p=2	(256, 32)	5,152
Max Pooling 1D	k=2, s=2	(128, 32)	–
Dropout	p=0.3	(8, 128)	–
Convolution 1D	k=3, s=1, p=1	(128, 64)	6,208
Convolution 1D	k=5, s=2, p=2	(64, 64)	20,544
Max Pooling 1D	k=2, s=2	(32, 64)	–
Dropout	p=0.3	(8, 128)	–
Convolution 1D	k=3, s=1, p=1	(32, 128)	24,704
Convolution 1D	k=5, s=2, p=2	(16, 128)	82,048
Max Pooling 1D	k=2, s=2	(8, 128)	–
Dropout	p=0.3	(8, 128)	–
Adaptive Avg Pool 1D	output=1	(1, 128)	–
Reshape	–	(128,)	–
Fully Connected	–	(8,)	1,032
ReLU	–	(8,)	–
Dropout	p=0.3	(8,)	–

Layer	Hyperparameters	Output Shape	Parameters
Fully Connected	–	(2,)	18
Softmax	–	(2,)	–

3.3.2.3 Simple 2D CNN

The *Simple2d* model is a small two-dimensional CNN and aims to test how the two-dimensional encoding of the input affects model performance. The input size is 32x16, which is the result of the 2D encoding of a 512-byte sequence. The first layer is a 1x1 convolution layer, bringing the filter space dimensionality from 1 to 128 while keeping the spatial dimensions. The rationale for this layer is to align the feature space with the embedding model introduced in Section 3.3.2.4. Then, the model consists of two convolutional blocks, each with two convolutional layers and a max pooling layer. After the convolutional blocks comes a fully-connected block with a single hidden layer for classification. Dropout with a rate of 0.3 is applied after each convolution block and between the two fully-connected layers. The full model specification is shown in Table 3.8. The model has a total of 184,794 trainable parameters and is hereby referred to as *Simple2d*.

Table 3.8: Simple 2D CNN

Layer	Hyperparameters	Output Shape	Parameters
Input	–	(32, 16, 1)	–
Convolution 2D	k=1, s=1, p=0	(32, 16, 128)	256
Dropout	p=0.3	(32, 16, 128)	–
Convolution 2D	k=3, s=1, p=1	(32, 16, 32)	36,896
Convolution 2D	k=5, s=2, p=2	(16, 8, 32)	25,632
Max Pooling 2D	k=2, s=2	(8, 4, 32)	–
Dropout	p=0.3	(8, 4, 32)	–
Convolution 2D	k=3, s=1, p=1	(8, 4, 64)	18,496
Convolution 2D	k=5, s=2, p=2	(4, 2, 64)	102,464
Max Pooling 2D	k=2, s=2	(2, 1, 64)	–
Dropout	p=0.3	(2, 1, 64)	–
Reshape	–	(128,)	–
Fully Connected	–	(8,)	1,032
ReLU	–	(8,)	–
Dropout	p=0.3	(8,)	–
Fully Connected	–	(2,)	18
Softmax	–	(2,)	–

3.3.2.4 Simple 2D CNN with embedding layer

Our two-dimensional embedding model builds on the simple 2D CNN model in Section 3.3.2.3 by placing an embedding layer at the beginning of the model instead of the 1x1 convolution layer.

The embedding layer transforms the byte values into a vector of continuous numbers, allowing the model to learn the characteristics of each byte value and represent it mathematically. After the embedding layer, the model is identical to the *Simple2d* model. The full model specification is shown in Table 3.9. This model has a total of 217,306 trainable parameters and is hereby referred to as *Simple2d-E*.

Table 3.9: Simple 2D CNN with embedding layer

Layer	Hyperparameters	Output Shape	Parameters
Input	–	(512,)	–
Embedding	v=256, d=128	(512, 128)	32,768
Dropout	p=0.3	(512, 128)	–
Reshape	–	(32, 16, 128)	–
Convolution 2D	k=3, s=1, p=1	(32, 16, 32)	36,896
Convolution 2D	k=5, s=2, p=2	(16, 8, 32)	25,632
Max Pooling 2D	k=2, s=2	(8, 4, 32)	–
Dropout	p=0.3	(8, 4, 32)	–
Convolution 2D	k=3, s=1, p=1	(8, 4, 64)	18,496
Convolution 2D	k=5, s=2, p=2	(4, 2, 64)	102,464
Max Pooling 2D	k=2, s=2	(2, 1, 64)	–
Dropout	p=0.3	(2, 1, 64)	–
Reshape	–	(128,)	–
Fully Connected	–	(8,)	1,032
ReLU	–	(8,)	–
Dropout	p=0.3	(8,)	–
Fully Connected	–	(2,)	18
Softmax	–	(2,)	–

3.3.2.5 ResNet50

ResNet is a common CNN architecture that utilizes *residual blocks*, which are groups of convolutional layers with skip connections [99]. ResNet comes in several variants, and we choose to use the variant with 50 weighted layers, commonly referred to as ResNet50.

The overall architecture of ResNet50 has:

- 1 convolutional layer
- 16 residual blocks, each containing 3 convolutional layers
- 1 average pooling layer
- 1 fully connected layer

To preprocess our data for ResNet50, we use the 2D image encoding described in Section 3.3.1.1, with a 32x32 image size. However, since ResNet expects a three-channel (RGB) image, we duplicate the pixel values to all three channels, which essentially results in a grayscale image. The

ResNet50 model from the PyTorch Torchvision library is used. It has a total of 23,512,130 parameters.

3.3.2.6 ResNet50 with embedding layer

This model architecture builds on the ResNet50 model described in Section 3.3.2.5, but modifies it to include an initial embedding layer. Specifically, the following modifications are made to the standard ResNet50 model:

- Added an embedding layer with vocabulary size 256 and dimension size 128 as the first layer
- Modified the first convolution layer to accept 128 channels instead of 3

The model takes a vector of length 1024 as input, which is reshaped to 32x32 after the embedding layer. The embedding layer increases the size of ResNet50 by a small factor, resulting in a model with 23,936,898 parameters. This model is hereby referred to as *ResNet50-E*.

3.3.3 Target features

For every model architecture, we will separately train and evaluate the model using these two target features:

- **Endianness** – the ordering of bytes in a multi-byte value.
- **Instruction width type** – whether the length of each instruction is fixed or variable.

We choose these features due to their importance in a reverse engineering process – if the reverse engineer can predictably split up the file into instructions of fixed width, it provides a solid starting point for deciphering instruction boundaries. Knowing the endianness allows the reverse engineer to properly interpret numerical values such as memory addresses.

Additionally, these features have certain technical properties that make them suitable for deep learning models. First, the features themselves have less ambiguous definitions than other ISA characteristics such as word size. Both features have clear classes that are well-suited for classification by models such as CNNs: the endianness of a file is typically either big or little, and the instruction width is either fixed or variable. Furthermore, the chosen features exhibit consistent byte patterns across the entire binary, allowing for analysis of small segments of binary code at a time.

3.4 Evaluation strategies

3.4.1 K-fold cross-validation on ISAdetect dataset

As an initial experiment, we use K-fold cross-validation (as described in Section 2.3.4.1) on the ISAdetect dataset to evaluate the performance of our models. We pick a value of 5 folds, a common choice for K-fold cross-validation that balances between computational cost and robustness. It should be noted that this experiment does not indicate how the model performs on previously unseen ISAs, since all 23 ISAs are present in the training data for each fold. However, we still include this experiment for the sake of interpreting the general behavior of the models.

3.4.2 Leave-one-group-out cross-validation on ISAdetect dataset

Our goal is to develop a CNN model that is able to discover features from binary executables of unseen ISAs. To validate whether our model generalizes to ISAs not present in the training data, we use Leave-One-Group-Out Cross-Validation (LOGO CV), using the ISAs as the groups (see Section 2.3.4.2 for a description of LOGO CV). In other words, we train models for validation using binaries from 22 out of our 23 ISAs from the ISAdetect dataset, using the single held-out group as the validation set.

Since LOGO CV trains a distinct model for each fold (one for each held-out ISA), we initialize each model with the same random seed across all 23 folds. This ensures identical starting weights, allowing us to attribute performance differences across folds to the difficulty in classifying each held-out ISA rather than to variations in initial conditions. We can then average performance metrics across folds to obtain a measure of how well that particular model configuration generalizes to unseen ISAs.

3.4.3 Testing on other datasets

To conduct further performance evaluation on ISAs not present in ISAdetect, we use the CpuRec dataset (described in Section 3.1.1.2) as well as BuildCross, the dataset we developed ourselves (described in Section 3.2). These evaluation strategies follow a train-test format, where we train the models on designated data from a training dataset, and run inference and test model performance on a testing dataset. Evaluating on additional datasets ensures comprehensive validation of model performance on a more diverse set of ISAs. Our choice of evaluation strategies is based on these factors:

- Focus on testing ISAs not present in the training set
- Alignment with the size and quality of the available datasets
- Limiting the number of target features, model variations, and evaluation strategies to ensure proper evaluation of the models within the time and resource constraints of this thesis

Table 3.10 shows the three evaluation strategies in which we use the additional datasets. For ease of reference, these evaluation strategies are hereby named *ISAdetect-CpuRec*, *ISAdetect-BuildCross*, and *Combined-CpuRec*.

Table 3.10: Evaluation strategies using multiple datasets .

Reference	Training dataset	Testing dataset
ISAdetect-CpuRec	ISAdetect	CpuRec
ISAdetect-BuildCross	ISAdetect	BuildCross
Combined-CpuRec	ISAdetect + BuildCross	CpuRec

3.4.4 Cross-seed validation

To account for the stochastic nature of deep neural network training, we validate each architecture by training multiple times with different seeds. The seed impacts factors such as weight initialization and data shuffling. By training using different seeds and averaging the performance metrics, we achieve a more reliable assessment of model performance by mitigating fortunate or unfortunate random initializations. Furthermore, we analyze the stability of our model architecture by examining the deviations in results across different initializations.

For the cross-validation evaluation strategies, we repeat the experiments 10 times with different seeds. For the experiments where we test on CpuRec and BuildCross, we repeat the experiments 20 times. To ensure reproducibility, the seeds and run configurations are documented in the source code repository [73].

3.4.5 Performance metrics and confidence intervals

To quantify the uncertainty in our model performance metrics, we calculate confidence intervals around the accuracy of our models. The confidence interval is a range of values that is likely to contain the true value of a parameter with a certain level of confidence, and the general statistical procedure is described in Section 2.3.6.1. In our case, we apply a 95% confidence interval for the mean accuracy of our models across multiple runs. We also calculate the mean and standard deviation of the per-ISA accuracies. Confusion matrices are used to identify systematic misclassifications for the different target features.

Chapter 4

Results

This chapter presents the results of the experiments described in Chapter 3. In Section 4.1, we present the results of applying our proposed model architectures for detecting endianness from a binary file. In Section 4.2, we present the results of applying the same model architectures for detecting instruction width type, that is, whether a binary file has fixed-length or variable-length instructions.

4.1 Endianness

This section evaluates and compares the performance of the proposed Convolutional Neural Network (CNN) models (described in Section 3.3.2) in detecting the endianness of binary files.

4.1.1 K-fold cross-validation on ISAdetect

In this experiment, we train and evaluate our models using 5-fold cross-validation as detailed in Section 3.4.1. Figure 4.1 shows the classification performance for every model. We see that all models achieve an average accuracy of above 99%.

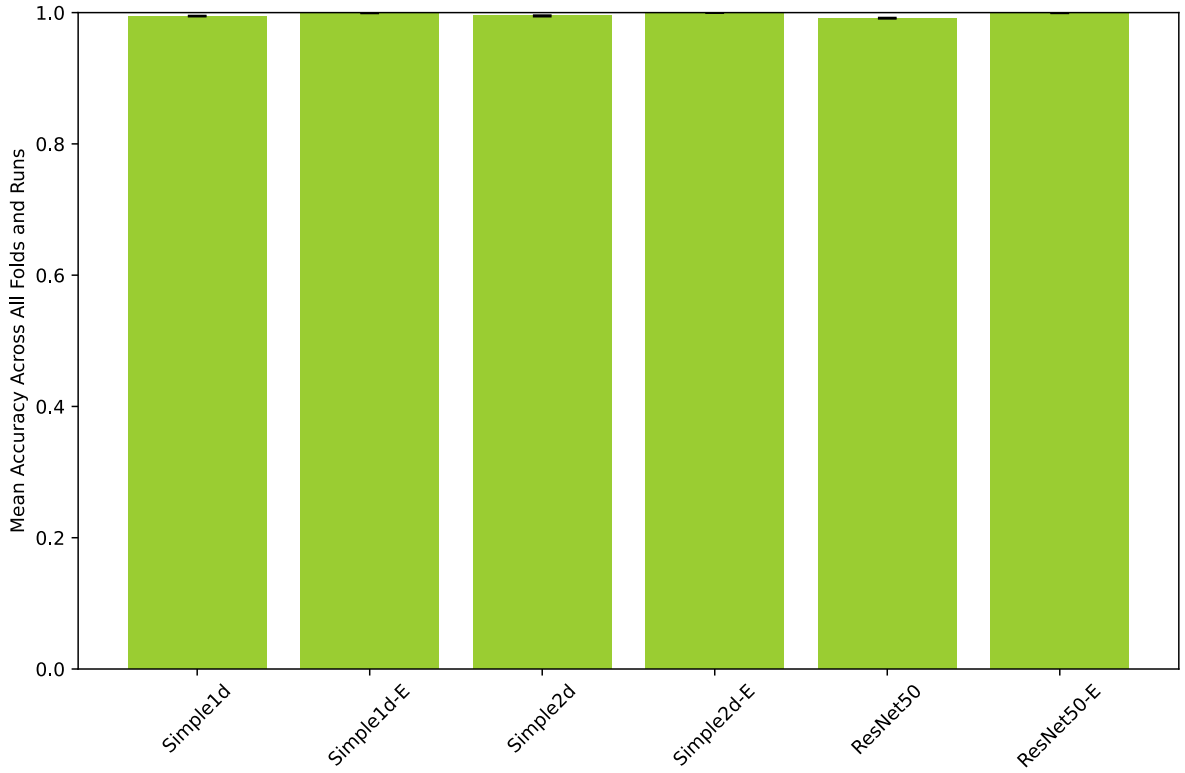


Figure 4.1: Endianness classification performance by model when using K-fold cross-validation on the ISAdetect dataset. Error bars indicate 95% confidence interval around the mean.

4.1.2 Leave-one-group-out cross-validation on ISAdetect

In this experiment, we train and evaluate our models using Leave-One-Group-Out Cross-Validation (LOGO CV) as detailed in Section 3.4.2. Table 4.1 shows classification performance for every model/ISA combination.

Table 4.1: Endianness classification performance when using LOGO CV on the ISAdetect dataset, across 10 runs. The reported performance numbers for each architecture are mean accuracy ± standard deviation. Overall mean accuracy and standard deviation across all runs are reported in bold at the bottom, along with a 95% confidence interval range around the mean.

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
alpha	0.997±0.004	0.953±0.043	1.000±0.000	0.938±0.073	0.973±0.029	0.990±0.011
amd64	0.981±0.011	0.987±0.020	0.943±0.016	0.991±0.022	0.952±0.014	0.993±0.013
arm64	0.713±0.291	0.730±0.259	0.413±0.098	0.545±0.297	0.251±0.096	0.562±0.316
armel	1.000±0.000	0.987±0.022	1.000±0.000	0.984±0.031	1.000±0.000	0.994±0.004
armhf	0.939±0.038	0.985±0.011	0.982±0.008	0.980±0.018	0.972±0.004	0.963±0.016
hppa	0.305±0.243	0.697±0.309	0.125±0.038	0.666±0.279	0.521±0.195	0.306±0.213
i386	0.969±0.028	0.998±0.003	0.980±0.014	0.997±0.005	0.970±0.013	0.992±0.005
ia64	0.414±0.428	0.884±0.058	0.098±0.236	0.611±0.414	0.482±0.308	0.372±0.369
m68k	0.008±0.015	0.096±0.200	0.000±0.000	0.008±0.020	0.028±0.050	0.036±0.076
mips	0.263±0.254	0.959±0.041	0.398±0.140	0.913±0.140	0.201±0.078	0.749±0.140
mips64el	0.994±0.005	0.998±0.002	1.000±0.000	0.996±0.009	0.935±0.056	0.991±0.017
mipsel	0.913±0.012	0.999±0.001	0.731±0.096	0.999±0.002	0.257±0.064	0.969±0.066

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
powerpc	0.999±0.002	1.000±0.000	0.999±0.001	1.000±0.000	0.997±0.002	1.000±0.000
powerpcspe	0.993±0.004	1.000±0.000	0.981±0.022	1.000±0.000	0.975±0.010	1.000±0.000
ppc64	0.869±0.202	0.816±0.196	0.931±0.109	0.769±0.200	0.627±0.220	0.392±0.277
ppc64el	1.000±0.000	1.000±0.000	1.000±0.001	1.000±0.000	0.828±0.158	0.988±0.017
riscv64	0.436±0.139	0.821±0.209	0.225±0.157	0.616±0.358	0.387±0.108	0.510±0.291
s390	0.070±0.210	0.999±0.002	0.005±0.004	0.992±0.015	0.205±0.354	0.716±0.360
s390x	0.389±0.294	1.000±0.000	0.064±0.095	0.995±0.014	0.332±0.312	0.762±0.229
sh4	0.273±0.233	0.928±0.144	0.138±0.202	0.832±0.259	0.331±0.242	0.740±0.259
sparc	0.994±0.009	0.999±0.001	0.665±0.253	0.998±0.002	0.706±0.118	0.900±0.095
sparc64	0.962±0.020	0.976±0.011	0.732±0.058	0.946±0.024	0.306±0.086	0.785±0.135
x32	0.990±0.006	0.999±0.002	0.983±0.006	0.999±0.001	0.977±0.005	1.000±0.000
Overall	**0.694±0.038**	**0.903±0.026**	**0.598±0.023**	**0.855±0.048**	**0.614±0.025**	**0.765±0.031**
95% CI	**0.665–0.723**	**0.883–0.923**	**0.581–0.615**	**0.819–0.891**	**0.595–0.634**	**0.741–0.789**

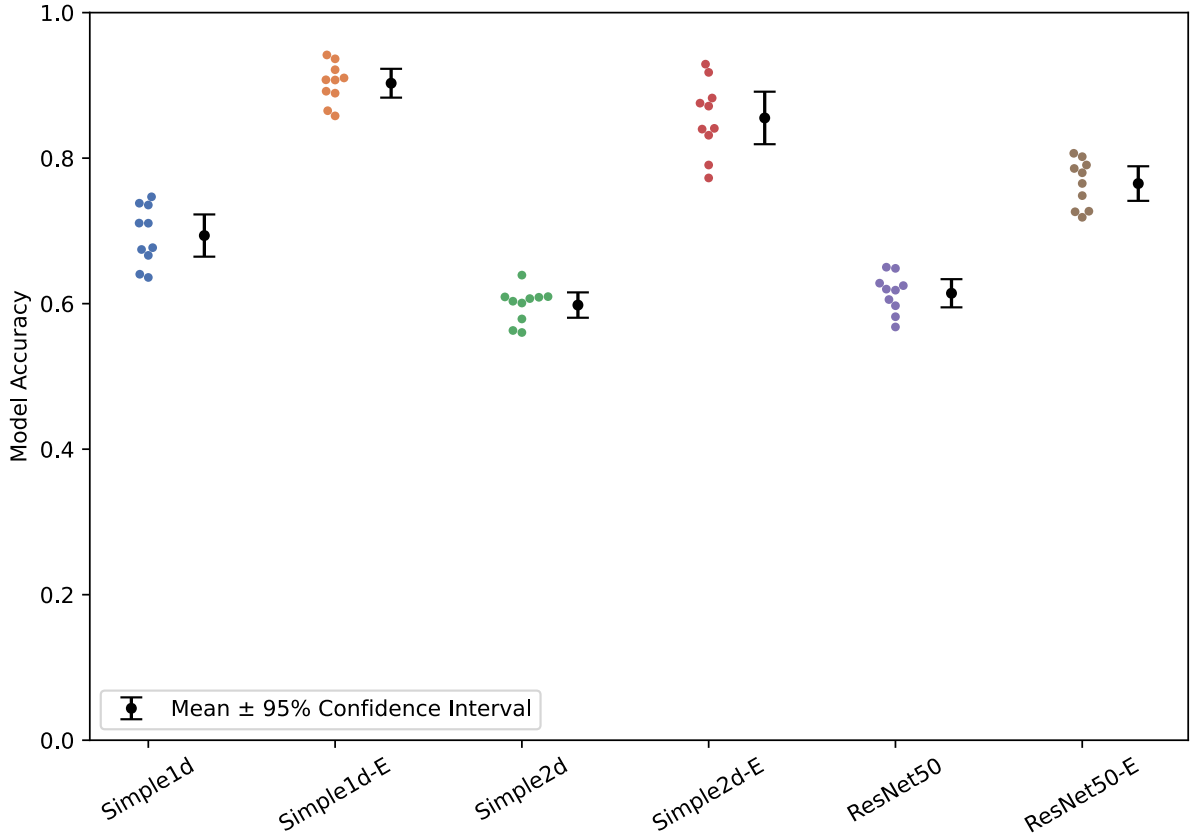


Figure 4.2: Endianness classification performance by model when using LOGO CV on the ISAdetect dataset. The data points in the swarm plot represents the model's overall accuracy for each of the 10 runs.

Figure 4.2 aggregates the results across ISAs, allowing for comparison of the overall performance of each model. We see that the *Simple1d-E* model performs the best, with a mean overall accuracy of 90.3%.

Figure 4.3 aggregates results across the different models, painting a picture of which ISAs are easier to classify. We observe that some architectures provide consistent performance across the

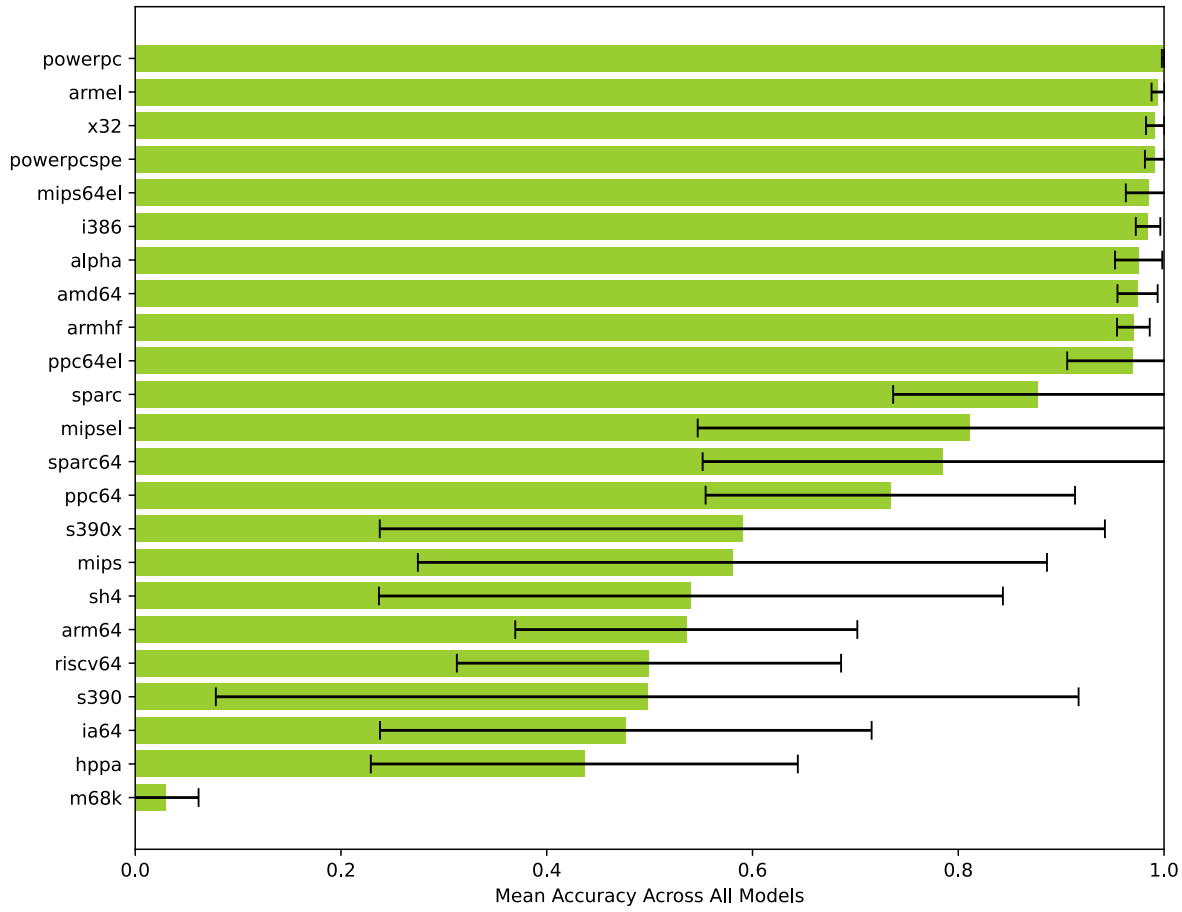


Figure 4.3: Endianness classification performance by Instruction Set Architecture (ISA) when using LOGO CV on the ISAdetect dataset. The error bars indicate the standard deviation across runs.

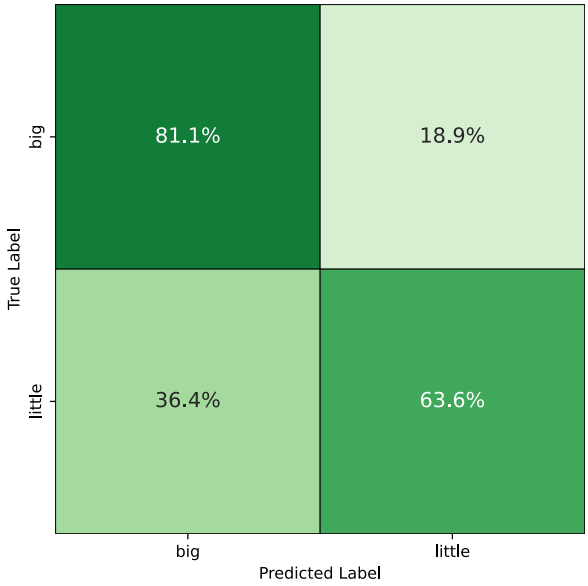


Figure 4.4: Confusion matrix of endianness classification when using LOGO CV on the ISAdetect dataset, aggregated across all models

models, while others exhibit extremely high variance across the different model architectures and seeds. Lastly, the `m68k` binaries are systematically misclassified by every model architecture.

Figure 4.4 aggregates the results across ISAs and models, and shows the confusion matrix. We see that the models are slightly biased towards predicting big-endian.

4.1.3 Training on ISAdetect, testing on CpuRec

For the ISAdetect-CpuRec experiment, each model is trained on the ISAdetect dataset and then performance-tested using the CpuRec dataset. Table 4.2 shows classification performance for every model/ISA combination. For each ISA and model combination, we report the number of correct classifications (out of 20 runs) for the single file present in the dataset.

Table 4.2: Endianness classification performance when training on the ISAdetect dataset and testing on the CpuRec dataset, across 20 runs for each model. Each row shows the number of correctly classified files over the total number of classification attempts. Overall mean accuracy and standard deviation across all runs are reported in bold at the bottom, along with a 95% confidence interval range around the mean.

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
6502	20/20	18/20	20/20	20/20	14/20	15/20
68HC08	0/20	1/20	0/20	2/20	2/20	0/20
68HC11	0/20	6/20	1/20	5/20	0/20	3/20
Alpha	20/20	20/20	20/20	20/20	18/20	20/20
ARC32eb	0/20	2/20	0/20	1/20	0/20	0/20
ARC32el	20/20	0/20	20/20	3/20	16/20	12/20
ARcompact	16/20	12/20	13/20	12/20	17/20	17/20
ARM64	20/20	20/20	20/20	20/20	20/20	20/20
ARMeb	20/20	19/20	20/20	20/20	13/20	9/20
ARMel	20/20	18/20	20/20	20/20	20/20	20/20
ARMhf	18/20	20/20	19/20	19/20	17/20	18/20
AxisCris	20/20	18/20	20/20	17/20	18/20	15/20
Blackfin	20/20	16/20	20/20	18/20	20/20	19/20
CLIPPER	8/20	16/20	2/20	14/20	4/20	20/20
CompactRISC	20/20	18/20	14/20	20/20	15/20	15/20
Epiphany	15/20	0/20	18/20	0/20	8/20	5/20
FR-V	20/20	15/20	20/20	19/20	11/20	17/20
FR30	1/20	4/20	18/20	2/20	11/20	3/20
FT32	20/20	20/20	5/20	20/20	7/20	20/20
H8-300	0/20	5/20	1/20	7/20	5/20	4/20
HP-PA	11/20	20/20	2/20	20/20	19/20	20/20
IA-64	20/20	14/20	20/20	19/20	17/20	18/20
IQ2000	4/20	20/20	2/20	20/20	6/20	15/20
M32C	15/20	14/20	20/20	14/20	20/20	17/20
M32R	3/20	9/20	7/20	12/20	2/20	4/20
M68k	0/20	20/20	2/20	20/20	12/20	16/20
M88k	20/20	20/20	20/20	20/20	19/20	19/20
MCore	15/20	5/20	17/20	11/20	4/20	16/20
Mico32	20/20	20/20	20/20	20/20	17/20	18/20
MicroBlaze	20/20	18/20	20/20	20/20	13/20	20/20
MIPSeb	20/20	20/20	20/20	20/20	14/20	20/20
MIPSel	20/20	20/20	20/20	20/20	18/20	20/20

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
MMIX	20/20	19/20	20/20	20/20	19/20	15/20
MN10300	20/20	17/20	20/20	18/20	20/20	19/20
Moxie	3/20	17/20	12/20	15/20	5/20	4/20
MSP430	19/20	15/20	20/20	11/20	8/20	16/20
NDS32	20/20	8/20	16/20	7/20	19/20	20/20
NIOS-II	19/20	7/20	20/20	4/20	12/20	14/20
PIC24	20/20	20/20	9/20	20/20	8/20	19/20
PPCeb	20/20	20/20	20/20	20/20	20/20	20/20
PPCel	20/20	20/20	20/20	20/20	20/20	20/20
RISC-V	4/20	20/20	0/20	20/20	2/20	16/20
RL78	20/20	20/20	20/20	19/20	20/20	18/20
ROMP	7/20	12/20	0/20	11/20	5/20	10/20
RX	20/20	19/20	18/20	17/20	20/20	19/20
S-390	0/20	20/20	0/20	20/20	8/20	19/20
SPARC	3/20	20/20	16/20	20/20	10/20	20/20
Stormy16	20/20	18/20	20/20	20/20	10/20	19/20
SuperH	20/20	20/20	11/20	20/20	11/20	20/20
V850	20/20	20/20	19/20	20/20	19/20	20/20
VAX	2/20	19/20	19/20	17/20	20/20	20/20
Visium	20/20	7/20	8/20	6/20	8/20	7/20
X86	20/20	20/20	19/20	19/20	19/20	20/20
X86-64	20/20	20/20	12/20	20/20	18/20	20/20
XtensaEB	0/20	1/20	3/20	1/20	6/20	6/20
Z80	20/20	18/20	20/20	15/20	20/20	20/20
Overall	**0.717±0.024**	**0.754±0.039**	**0.699±0.028**	**0.763±0.048**	**0.646±0.045**	**0.764±0.040**
95% CI	**0.687–0.747**	**0.722–0.787**	**0.668–0.731**	**0.729–0.798**	**0.610–0.683**	**0.732–0.797**

Figure 4.5 aggregates the results across ISAs. We observe that the three embedding models perform the best, and the performance differences between the three embedding models are down to the margin of error.

Figure 4.6 aggregates results across the different models. Again, we observe that some architectures provide consistent performance across the models, while others exhibit extremely high variance across the different model architectures and seeds. The mean accuracy ranges from close to 0% for `ARC32eb` to 100% for `ARM64`, `PPCel`, and `PPCeb`.

Figure 4.7 aggregates the results across ISAs and models, and shows the confusion matrix. We see that the models are significantly biased towards predicting big-endian.

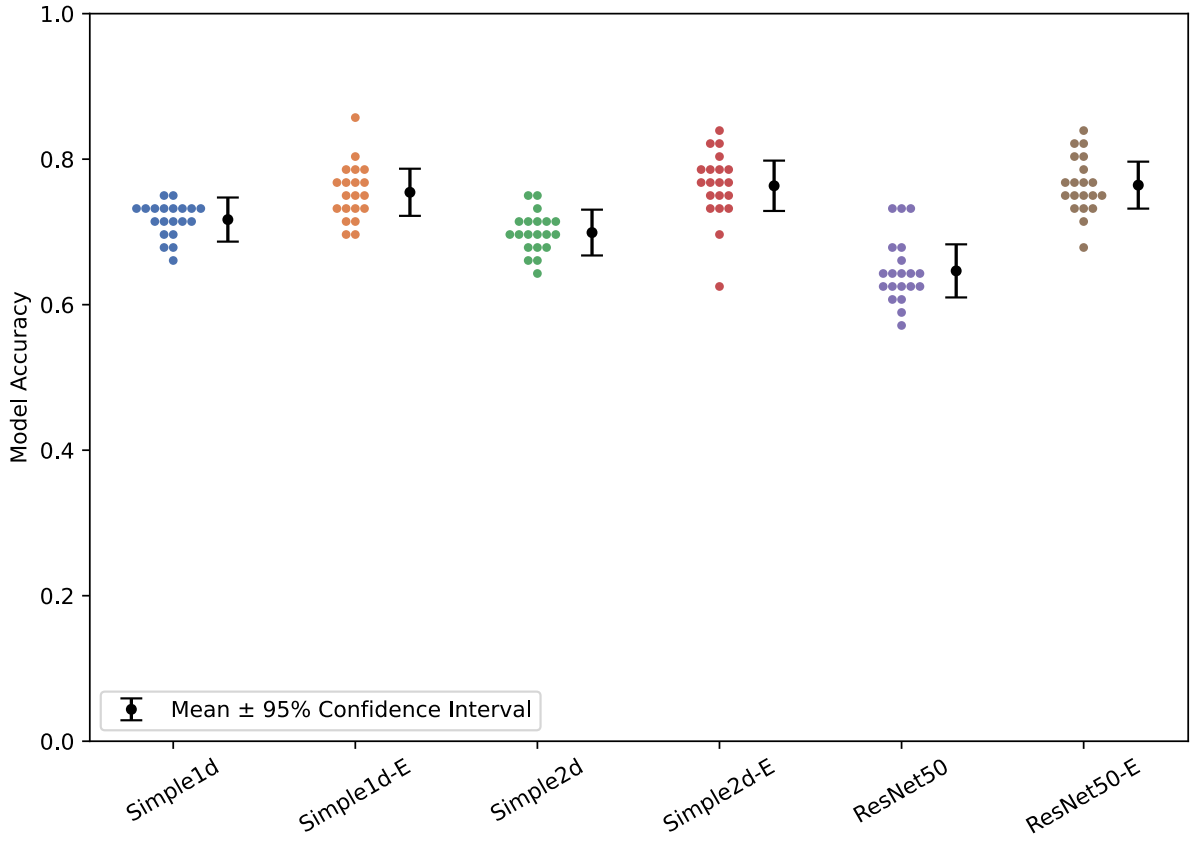


Figure 4.5: Endianness classification performance by model when training on the ISAdetect dataset and testing on the CpuRec dataset. The data points in the swarm plot represents the model's overall accuracy for each of the 20 runs.

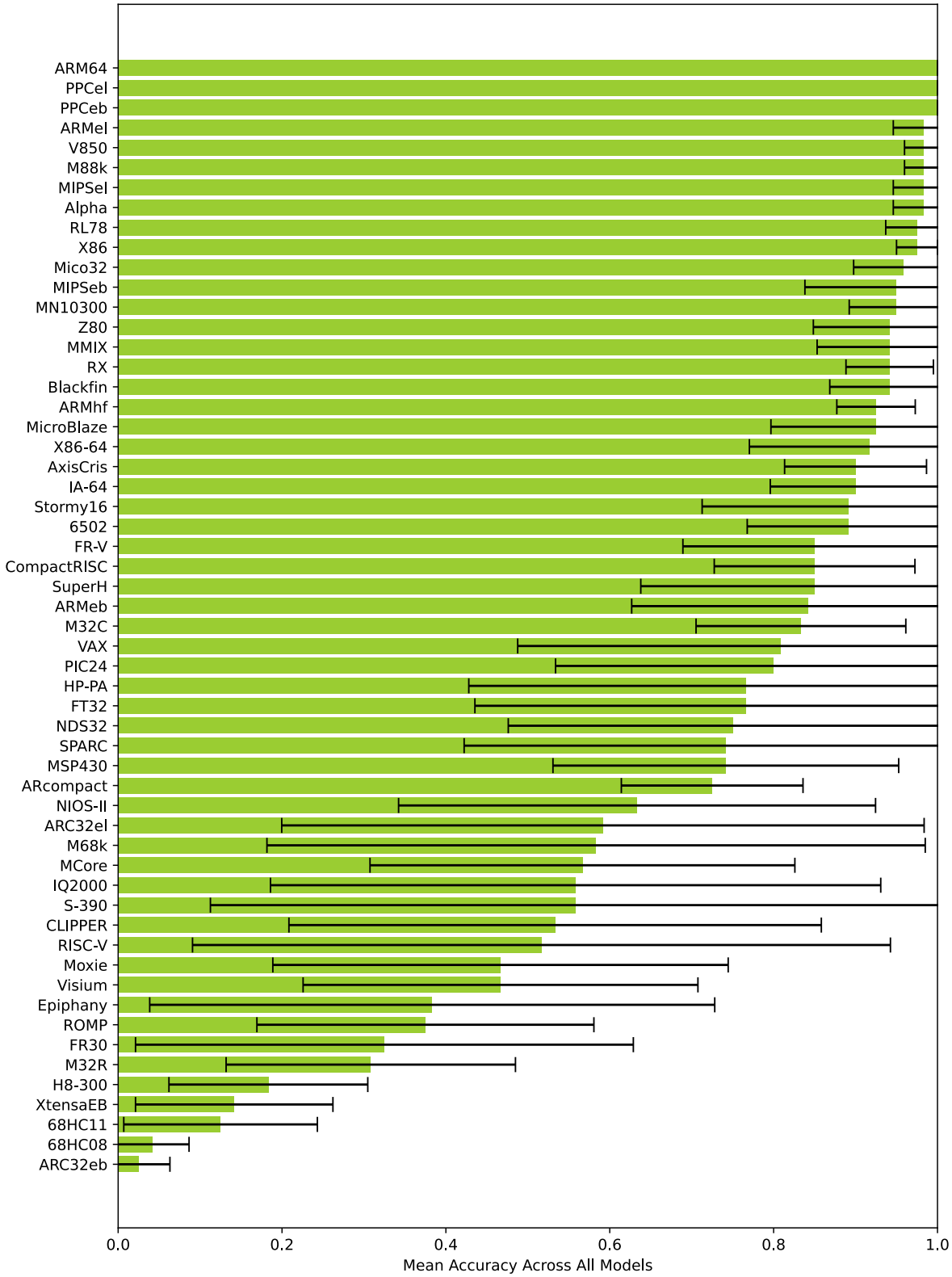


Figure 4.6: Endianness classification performance by ISA when training on the ISAdetect dataset and testing on the CpuRec dataset. The error bars indicate the standard deviation across runs.

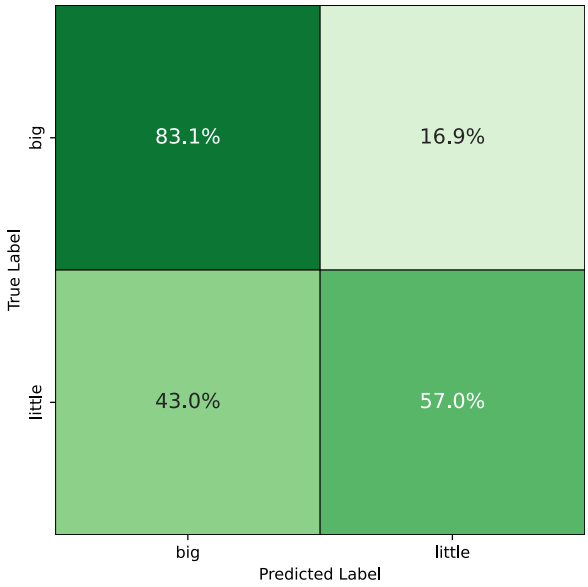


Figure 4.7: Confusion matrix of endianness classification when training on the ISAdetect dataset and testing on the CpuRec dataset, aggregated across all models

4.1.4 Training on ISAdetect, testing on BuildCross

For the ISAdetect-BuildCross experiment, each model is trained on the ISAdetect dataset and then performance-tested using the BuildCross dataset. Table 4.3 shows classification performance for every model/ISA combination.

Table 4.3: Endianness classification performance when training on the ISAdetect dataset and testing on the BuildCross dataset, across 20 runs. The reported performance numbers for each architecture are mean accuracy ± standard deviation. Overall mean accuracy and standard deviation across all runs are reported in bold at the bottom, along with a 95% confidence interval range around the mean.

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
arc	0.910±0.048	0.358±0.272	0.873±0.067	0.379±0.256	0.764±0.078	0.688±0.221
arceb	0.032±0.027	0.065±0.046	0.067±0.036	0.056±0.043	0.238±0.044	0.133±0.130
bfin	0.940±0.046	0.742±0.148	0.918±0.044	0.720±0.165	0.811±0.038	0.813±0.132
bpf	0.005±0.014	0.825±0.276	0.180±0.124	0.812±0.305	0.576±0.146	0.713±0.292
c6x	0.114±0.129	0.949±0.135	0.002±0.003	0.991±0.014	0.185±0.111	0.517±0.285
cr16	0.935±0.044	0.379±0.137	0.839±0.071	0.424±0.173	0.470±0.078	0.679±0.149
cris	0.992±0.008	0.857±0.131	0.997±0.003	0.838±0.119	0.967±0.011	0.897±0.080
csky	0.817±0.105	0.168±0.121	0.890±0.057	0.208±0.217	0.576±0.115	0.368±0.191
epiphany	0.595±0.179	0.303±0.161	0.584±0.115	0.366±0.185	0.520±0.085	0.525±0.169
fr30	0.288±0.134	0.108±0.217	0.222±0.138	0.078±0.175	0.384±0.085	0.093±0.187
frv	0.897±0.069	0.862±0.183	0.951±0.020	0.841±0.192	0.780±0.076	0.608±0.243
ft32	0.985±0.029	0.150±0.220	0.997±0.005	0.097±0.116	0.648±0.118	0.569±0.281
h8300	0.165±0.128	0.171±0.249	0.236±0.096	0.213±0.279	0.167±0.050	0.027±0.042
iq2000	0.843±0.147	0.925±0.122	0.599±0.163	0.943±0.053	0.722±0.159	0.755±0.188
kvx	0.812±0.109	0.438±0.191	0.824±0.095	0.571±0.183	0.667±0.095	0.655±0.178
lm32	0.950±0.047	0.803±0.166	0.709±0.042	0.896±0.088	0.581±0.057	0.818±0.179
loongarch64	0.319±0.173	0.719±0.212	0.075±0.053	0.661±0.205	0.123±0.075	0.749±0.201
m32r	0.854±0.099	0.709±0.187	0.921±0.032	0.850±0.156	0.529±0.094	0.668±0.224
m68k-elf	0.205±0.122	0.727±0.134	0.199±0.080	0.665±0.140	0.329±0.055	0.471±0.141

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
mcore	0.959±0.031	0.649±0.287	0.874±0.086	0.617±0.318	0.222±0.084	0.773±0.217
mcoreeb	0.022±0.018	0.491±0.272	0.291±0.138	0.543±0.260	0.707±0.115	0.329±0.241
microblaze	0.974±0.045	0.861±0.140	0.997±0.002	0.935±0.065	0.897±0.049	0.931±0.062
microblazeel	0.973±0.030	0.719±0.160	0.999±0.001	0.754±0.058	0.951±0.040	0.901±0.053
mmix	0.839±0.146	0.668±0.260	0.914±0.081	0.719±0.181	0.757±0.121	0.528±0.228
mn10300	0.940±0.034	0.736±0.275	0.983±0.018	0.792±0.221	0.970±0.075	0.882±0.134
moxie	0.366±0.214	0.694±0.170	0.254±0.089	0.675±0.196	0.319±0.074	0.393±0.184
moxieel	0.623±0.203	0.864±0.154	0.645±0.115	0.919±0.073	0.718±0.077	0.943±0.050
msp430	0.688±0.153	0.852±0.187	0.865±0.087	0.878±0.099	0.549±0.194	0.946±0.053
nds32	0.791±0.123	0.506±0.226	0.431±0.168	0.528±0.188	0.439±0.120	0.668±0.151
nios2	0.897±0.105	0.191±0.147	0.971±0.017	0.189±0.180	0.916±0.023	0.530±0.302
or1k	0.954±0.036	0.519±0.272	0.718±0.090	0.675±0.283	0.461±0.118	0.411±0.229
pru	0.029±0.022	0.101±0.051	0.053±0.022	0.094±0.050	0.887±0.082	0.174±0.117
rl78	0.972±0.022	0.877±0.140	0.991±0.008	0.847±0.151	0.993±0.007	0.932±0.080
rx	0.966±0.018	0.857±0.152	0.923±0.047	0.837±0.149	0.929±0.027	0.885±0.125
tilegx	0.971±0.017	0.082±0.216	0.964±0.027	0.100±0.183	0.595±0.151	0.309±0.307
tricore	0.929±0.047	0.974±0.017	0.976±0.010	0.981±0.008	0.968±0.012	0.981±0.015
v850	0.778±0.107	0.902±0.105	0.540±0.137	0.860±0.162	0.475±0.137	0.958±0.051
visium	0.401±0.132	0.370±0.274	0.862±0.048	0.418±0.256	0.582±0.148	0.174±0.152
xstormy16	0.884±0.063	0.605±0.338	0.590±0.189	0.600±0.347	0.332±0.141	0.942±0.091
xtensa	0.067±0.042	0.270±0.183	0.160±0.094	0.256±0.129	0.556±0.089	0.179±0.127
Overall	**0.713±0.010**	**0.542±0.049**	**0.689±0.006**	**0.571±0.038**	**0.610±0.018**	**0.571±0.040**
95% CI	**0.708–0.717**	**0.520–0.565**	**0.687–0.692**	**0.553–0.589**	**0.602–0.619**	**0.552–0.589**

Figure 4.8 aggregates the results across ISAs. Here, contrary to the other testing setups, we observe that the embedding models are performing worse than their non-embedding counterparts.

Figure 4.9 aggregates results across the different models. We see that the average classification performance varies substantially across the different ISAs, from 10% for the `arceb` architecture to 96% for the `tricore` architecture.

Figure 4.10 aggregates the results across ISAs and models, and shows the confusion matrix. We see that even though overall classification performance is low, the models appear balanced and do not show any clear bias towards a specific class.

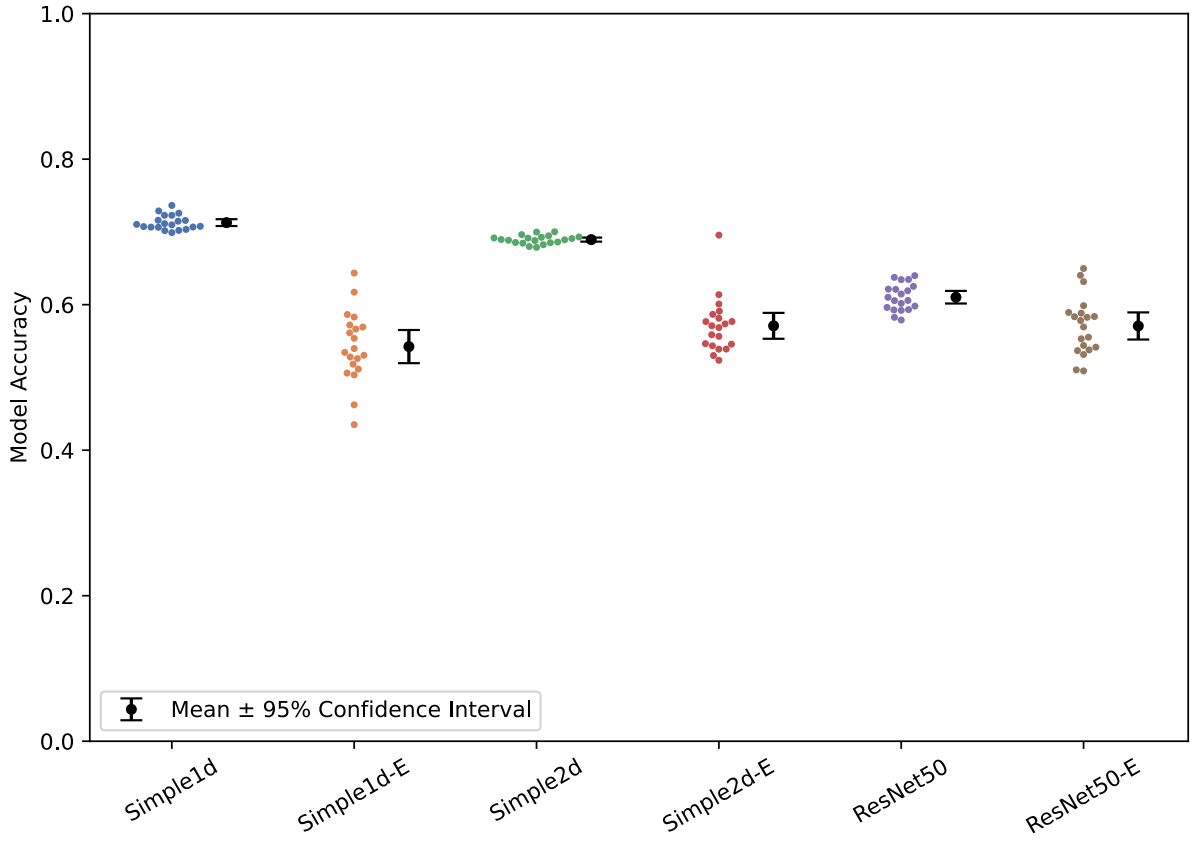


Figure 4.8: Endianness classification performance by model when training on the ISAdetect dataset and testing on the BuildCross dataset. The data points in the swarm plot represents the model's overall accuracy for each of the 20 runs.

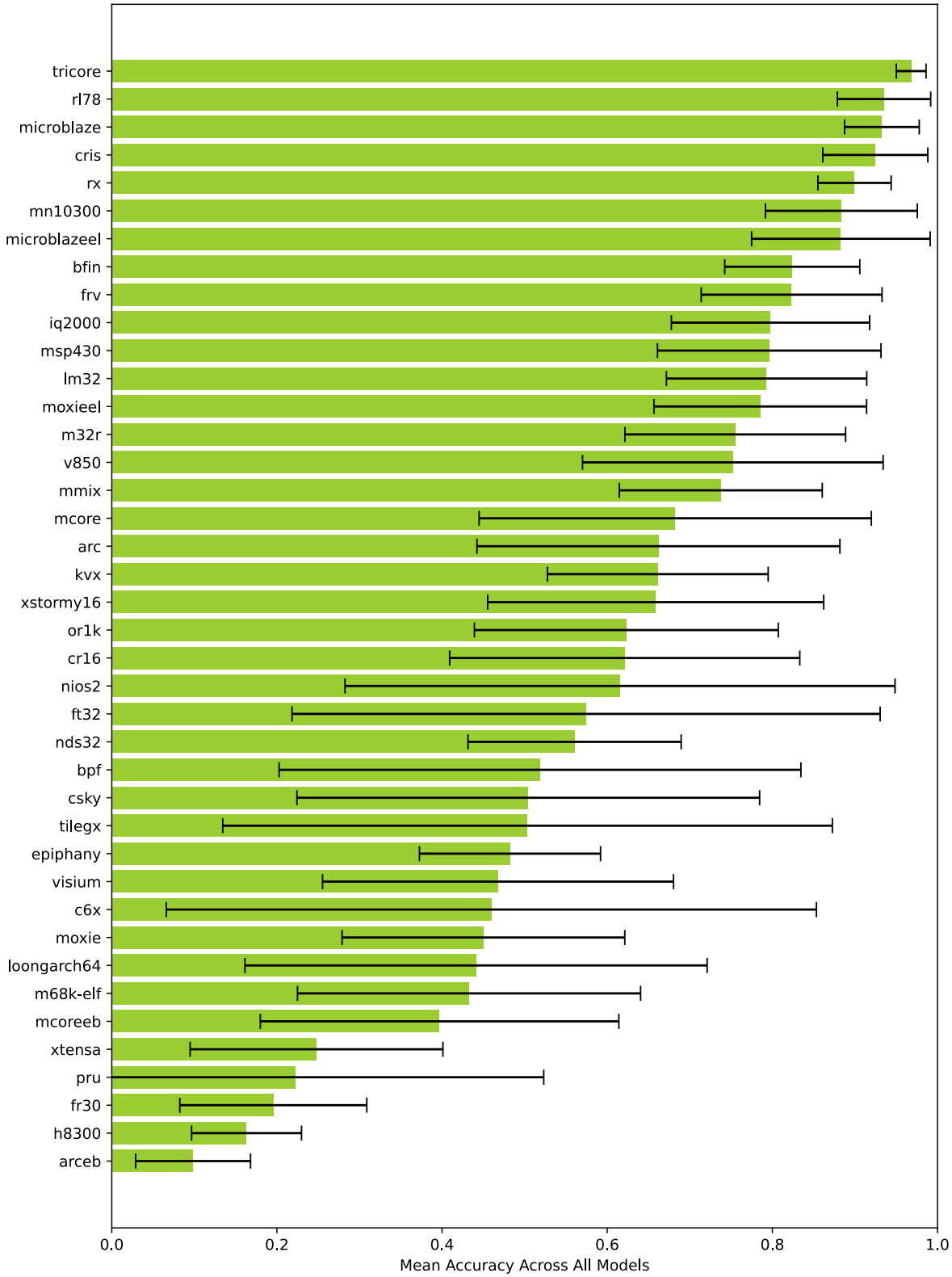


Figure 4.9: Endianness classification performance by ISA when training on the ISAdetect dataset and testing on the BuildCross dataset. The error bars indicate the standard deviation across runs.

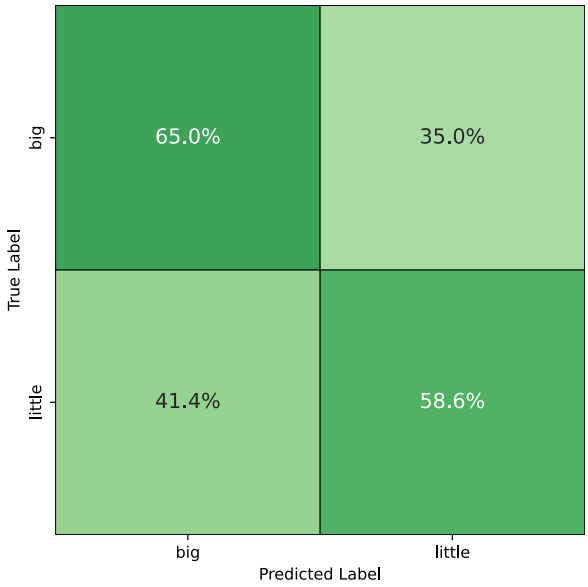


Figure 4.10: Confusion matrix of endianness classification when training on the ISAdetect dataset and testing on the BuildCross dataset, aggregated across all models

4.1.5 Training on ISAdetect and BuildCross, testing on CpuRec

For the Combined-CpuRec experiment, we train the models on the ISAdetect and BuildCross datasets, and test them on the CpuRec dataset. Table 4.4 shows classification performance for every model/ISA combination.

Table 4.4: Endianness classification performance when training on the ISAdetect and BuildCross datasets, and testing on the CpuRec dataset, across 20 runs for each model. Each row shows the number of correctly classified files over the total number of classification attempts. Overall mean accuracy and standard deviation across all runs are reported in bold at the bottom, along with a 95% confidence interval range around the mean.

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
6502	20/20	7/20	19/20	1/20	5/20	6/20
68HC08	0/20	7/20	0/20	6/20	0/20	5/20
68HC11	2/20	0/20	8/20	0/20	0/20	1/20
Alpha	20/20	20/20	20/20	20/20	19/20	20/20
ARC32eb	6/20	4/20	0/20	0/20	1/20	3/20
ARC32el	20/20	7/20	20/20	7/20	20/20	14/20
ARcompact	6/20	20/20	12/20	20/20	18/20	20/20
ARM64	20/20	19/20	20/20	19/20	14/20	20/20
ARMeb	18/20	20/20	16/20	20/20	20/20	17/20
ARMel	20/20	20/20	20/20	20/20	20/20	20/20
ARMhf	8/20	10/20	2/20	12/20	12/20	16/20
AxisCris	20/20	19/20	20/20	20/20	20/20	20/20
Blackfin	9/20	20/20	20/20	20/20	19/20	20/20
CLIPPER	17/20	18/20	20/20	20/20	19/20	19/20
CompactRISC	13/20	20/20	3/20	20/20	14/20	20/20
Epiphany	16/20	20/20	13/20	20/20	14/20	20/20
FR-V	20/20	20/20	20/20	20/20	20/20	20/20
FR30	0/20	20/20	7/20	20/20	12/20	20/20

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
FT32	2/20	12/20	15/20	20/20	16/20	19/20
H8-300	8/20	13/20	20/20	6/20	5/20	5/20
HP-PA	1/20	20/20	7/20	20/20	19/20	20/20
IA-64	18/20	17/20	20/20	16/20	15/20	19/20
IQ2000	19/20	19/20	1/20	14/20	13/20	17/20
M32C	17/20	20/20	20/20	16/20	17/20	17/20
M32R	19/20	20/20	4/20	19/20	6/20	16/20
M68k	1/20	20/20	6/20	20/20	0/20	20/20
M88k	20/20	20/20	20/20	20/20	20/20	20/20
MCore	17/20	19/20	19/20	18/20	8/20	15/20
Mico32	20/20	20/20	20/20	20/20	20/20	20/20
MicroBlaze	20/20	20/20	20/20	20/20	20/20	20/20
MIPSeb	20/20	20/20	18/20	20/20	20/20	20/20
MIPSel	20/20	20/20	20/20	20/20	20/20	20/20
MMIX	20/20	20/20	20/20	20/20	20/20	20/20
MN10300	20/20	18/20	20/20	20/20	19/20	20/20
Moxie	1/20	20/20	20/20	20/20	18/20	20/20
MSP430	20/20	20/20	17/20	20/20	14/20	20/20
NDS32	18/20	17/20	7/20	19/20	16/20	19/20
NIOS-II	3/20	20/20	20/20	20/20	13/20	20/20
PIC24	20/20	6/20	20/20	11/20	20/20	16/20
PPCeb	20/20	20/20	20/20	20/20	20/20	20/20
PPCel	20/20	20/20	20/20	20/20	20/20	20/20
RISC-V	0/20	18/20	1/20	16/20	5/20	7/20
RL78	20/20	16/20	19/20	18/20	18/20	12/20
ROMP	6/20	5/20	13/20	0/20	13/20	0/20
RX	12/20	20/20	19/20	20/20	19/20	20/20
S-390	0/20	20/20	8/20	20/20	10/20	18/20
SPARC	1/20	20/20	14/20	20/20	17/20	20/20
Stormy16	20/20	20/20	20/20	20/20	19/20	20/20
SuperH	19/20	20/20	4/20	20/20	9/20	20/20
V850	20/20	20/20	20/20	20/20	20/20	20/20
VAX	1/20	3/20	16/20	4/20	16/20	11/20
Visium	20/20	20/20	16/20	19/20	20/20	19/20
X86	20/20	20/20	17/20	20/20	20/20	20/20
X86-64	11/20	20/20	3/20	20/20	18/20	18/20
XtensaEB	7/20	2/20	7/20	2/20	2/20	18/20
Z80	20/20	20/20	20/20	20/20	20/20	20/20
Overall	**0.675±0.033**	**0.836±0.035**	**0.724±0.028**	**0.824±0.034**	**0.743±0.047**	**0.846±0.019**
95% CI	**0.642–0.708**	**0.807–0.864**	**0.693–0.755**	**0.796–0.853**	**0.708–0.778**	**0.821–0.870**

Figure 4.11 aggregates the results across ISAs. Notably, introducing the BuildCross dataset as additional training data improves classification performance for most models.

Figure 4.12 aggregates results across the different models.

Figure 4.13 aggregates the results across ISAs and models, and shows the confusion matrix. We see that the models are slightly biased towards predicting big-endian.

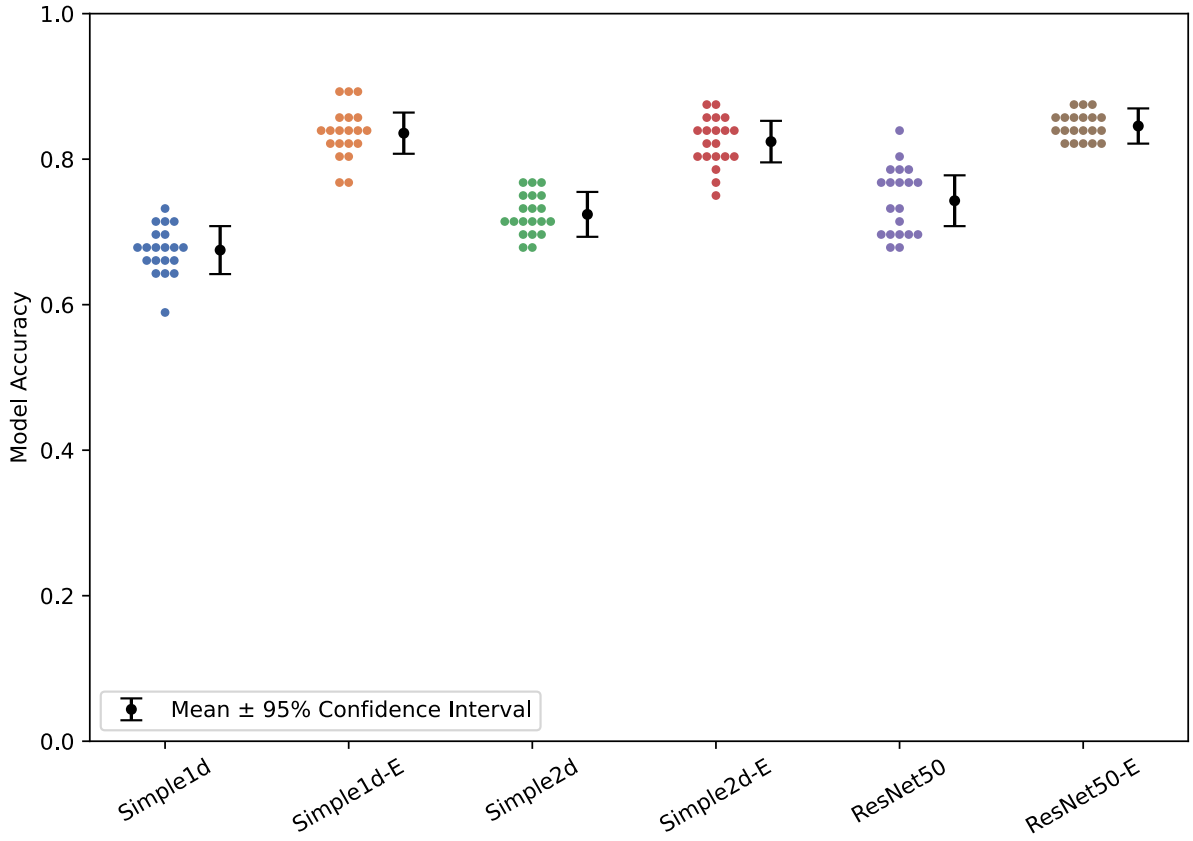


Figure 4.11: Endianness classification performance by model when training on the ISAdetect and BuildCross datasets, and testing on the CpuRec dataset. The data points in the swarm plot represents the model's overall accuracy for each of the 20 runs.

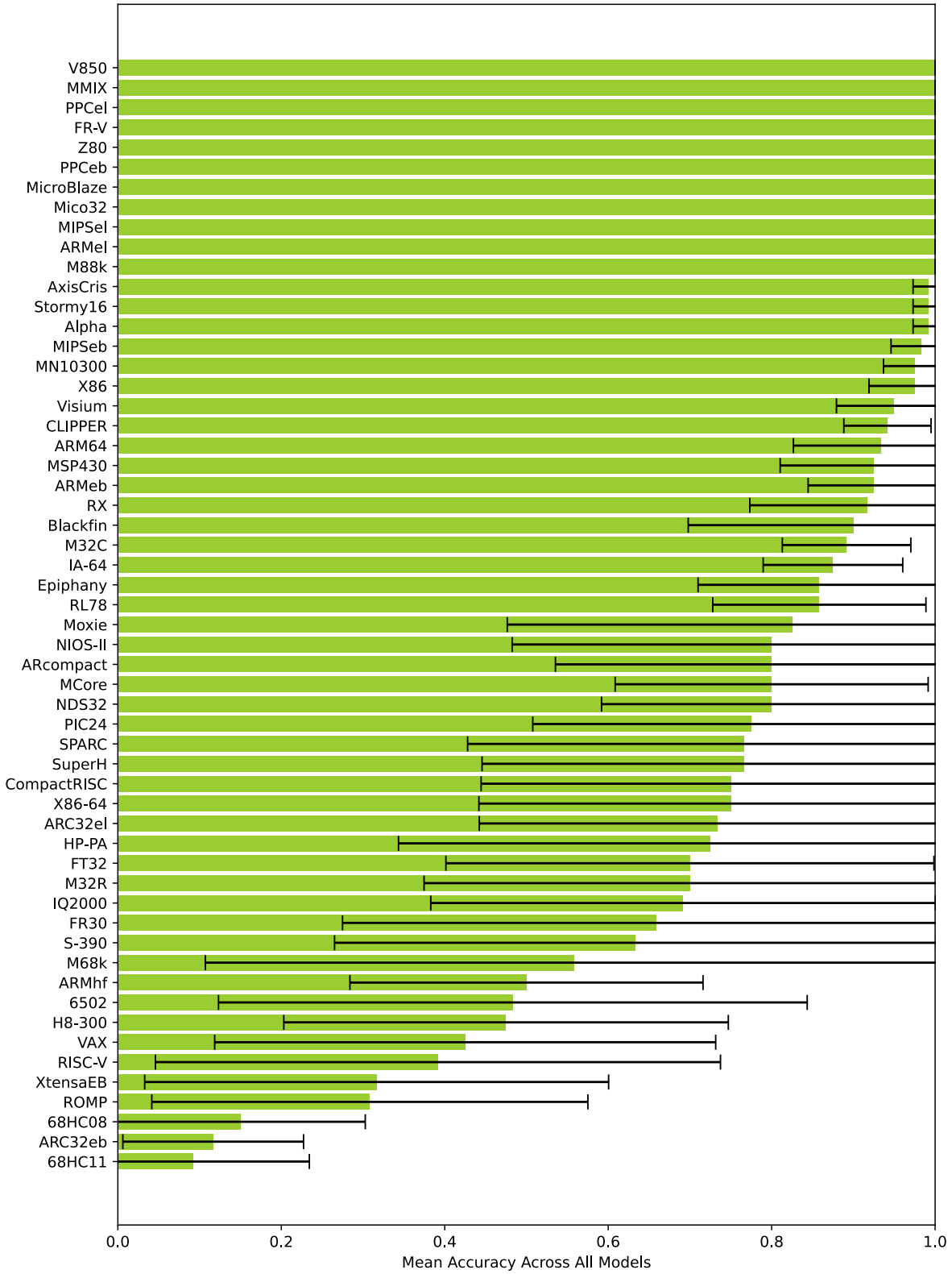


Figure 4.12: Endianness classification performance by ISA when training on the ISAdetect and BuildCross datasets, and testing on the CpuRec dataset. The error bars indicate the standard deviation across runs.

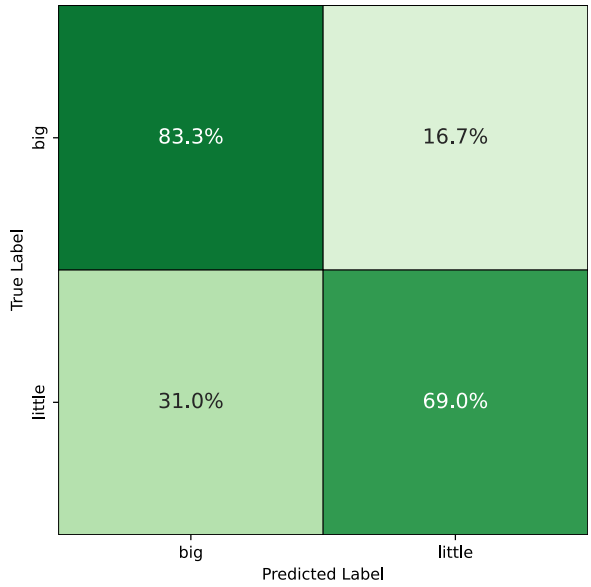


Figure 4.13: Confusion matrix of endianness classification when training on the ISAdetect and BuildCross datasets, and testing on the CpuRec dataset, aggregated across all models

4.2 Instruction width type

This section evaluates and compares the performance of the proposed CNN models (described in Section 3.3.2) in detecting the instruction width type of binary files.

4.2.1 K-fold cross-validation on ISAdetect

In this experiment, we train and evaluate our models using 5-fold cross-validation as detailed in Section 3.4.1. Figure 4.14 shows the classification performance for every model. We see that all models achieve an average accuracy of above 99%.

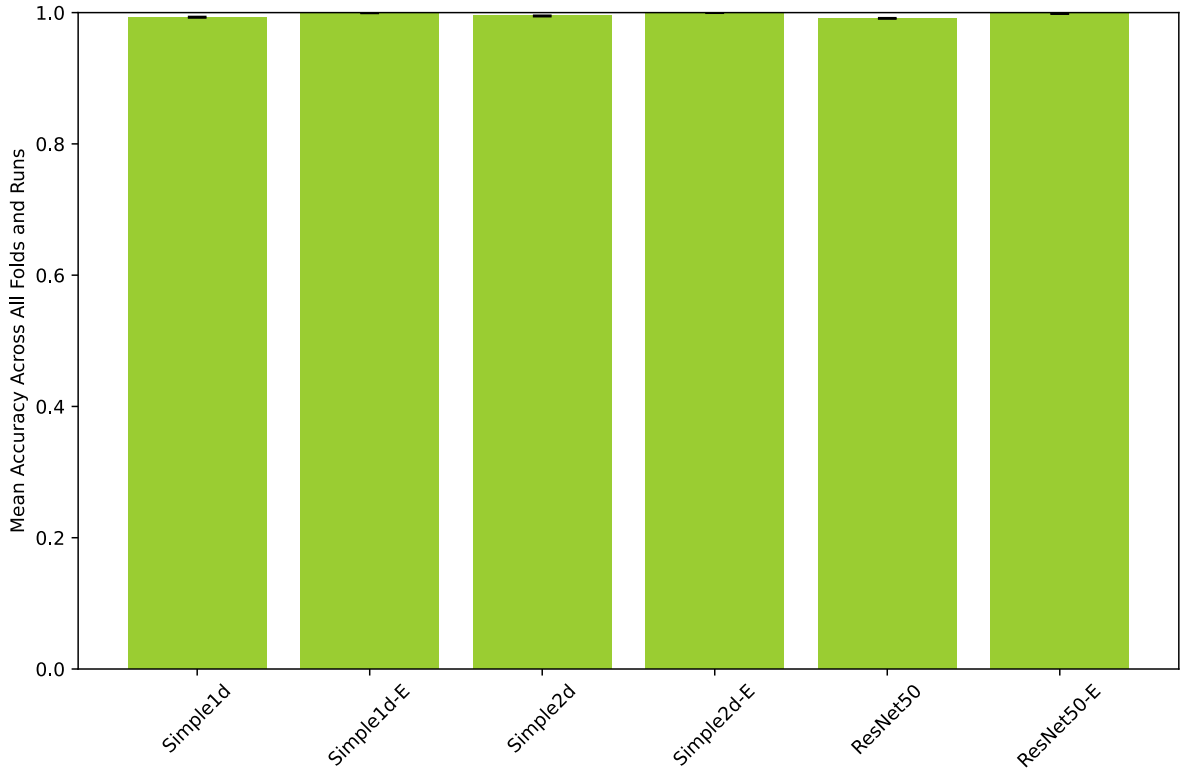


Figure 4.14: Instruction width type classification performance by model when using K-fold cross-validation on the ISAdetect dataset. Error bars indicate 95% confidence interval around the mean.

4.2.2 Training and testing on ISAdetect

In this experiment, we train and evaluate our models using LOGO CV as detailed in Section 3.4.2. Table 4.5 shows classification performance for every model/ISA combination.

Table 4.5: Instruction width type classification performance when using LOGO CV on the ISAdetect dataset, across 10 runs. The reported performance numbers for each architecture are mean accuracy ± standard deviation. Overall mean accuracy and standard deviation across all runs are reported in bold at the bottom, along with a 95% confidence interval range around the mean.

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
alpha	0.917±0.179	0.229±0.082	0.761±0.311	0.242±0.069	0.159±0.074	0.256±0.134
amd64	0.926±0.032	0.996±0.008	0.916±0.017	0.993±0.008	0.870±0.015	0.985±0.008
arm64	0.911±0.075	0.520±0.314	0.996±0.005	0.494±0.364	0.977±0.012	0.957±0.056
armel	1.000±0.000	0.994±0.004	1.000±0.000	0.995±0.004	0.996±0.003	0.986±0.011
armhf	0.997±0.004	0.981±0.012	0.938±0.005	0.981±0.008	0.953±0.013	0.974±0.013
hppa	0.299±0.168	0.778±0.268	0.236±0.084	0.762±0.289	0.944±0.082	0.869±0.207
i386	0.892±0.075	0.997±0.003	0.881±0.096	0.997±0.002	0.820±0.026	0.929±0.032
ia64	0.010±0.002	0.698±0.371	0.588±0.419	0.088±0.274	0.004±0.007	0.225±0.301
m68k	0.984±0.033	0.839±0.331	0.261±0.399	0.928±0.217	0.571±0.417	0.905±0.172
mips	0.990±0.014	0.861±0.242	0.997±0.003	0.836±0.189	0.994±0.007	0.946±0.062
mips64el	0.990±0.008	0.980±0.056	0.997±0.003	0.921±0.137	0.873±0.084	0.970±0.047
mipsel	0.928±0.052	0.924±0.117	0.993±0.009	0.952±0.062	0.987±0.009	0.995±0.006
powerpc	1.000±0.000	1.000±0.000	1.000±0.000	1.000±0.000	0.999±0.001	1.000±0.000

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
powerpcspe	0.999±0.002	1.000±0.000	1.000±0.000	1.000±0.000	0.995±0.006	1.000±0.000
ppc64	0.946±0.058	0.626±0.270	0.795±0.139	0.336±0.177	0.891±0.063	0.704±0.178
ppc64el	0.219±0.319	0.849±0.330	0.787±0.391	0.735±0.318	0.978±0.019	0.998±0.002
riscv64	0.117±0.059	0.994±0.020	0.810±0.226	1.000±0.001	0.889±0.094	0.996±0.007
s390	0.580±0.202	0.993±0.009	0.048±0.021	0.979±0.034	0.205±0.289	0.344±0.398
s390x	0.074±0.100	0.986±0.028	0.068±0.167	0.996±0.004	0.603±0.321	0.490±0.372
sh4	0.846±0.229	0.895±0.204	0.965±0.106	0.999±0.001	0.982±0.029	0.999±0.001
sparc	0.980±0.044	1.000±0.000	0.384±0.049	1.000±0.000	0.882±0.101	0.999±0.001
sparc64	0.861±0.166	0.969±0.020	0.804±0.102	0.984±0.013	0.918±0.075	0.983±0.029
x32	0.977±0.008	0.999±0.002	0.984±0.006	0.998±0.002	0.968±0.005	0.995±0.003
Overall	**0.750±0.029**	**0.880±0.040**	**0.734±0.051**	**0.843±0.026**	**0.789±0.031**	**0.843±0.036**
95% CI	**0.728–0.772**	**0.850–0.911**	**0.696–0.773**	**0.823–0.863**	**0.765–0.812**	**0.817–0.870**

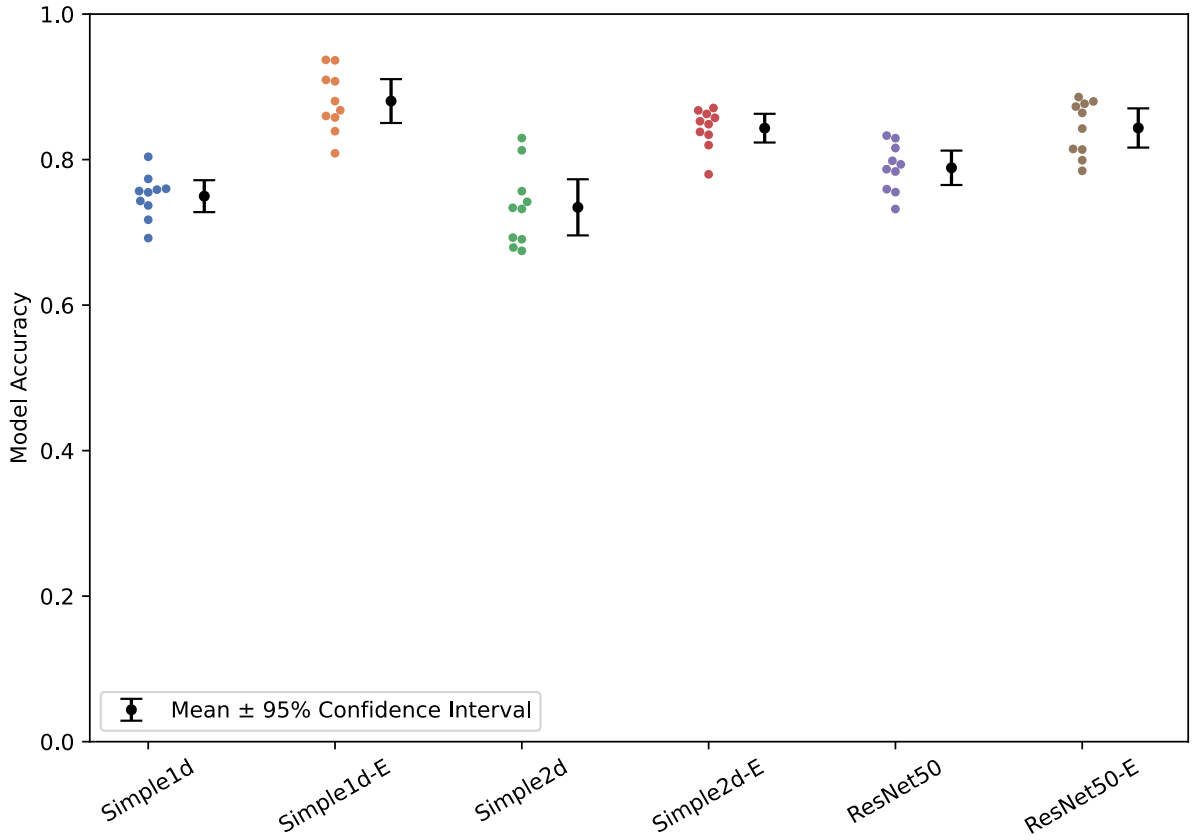


Figure 4.15: Instruction width type classification performance by model when using LOGO CV on the ISAdetect dataset. The data points in the swarm plot represents the model's overall accuracy for each of the 10 runs.

Figure 4.15 aggregates the results across ISAs. Similar to the performance seen with endianness classification, the embedding models perform better than the non-embedding counterparts, with the *Simple1d-E* model performing the best with a mean accuracy of 88.0%.

Figure 4.16 aggregates results across the different models. Many ISAs are correctly classified with accuracy well above 90%, while others perform poorly and show large performance variations across runs.

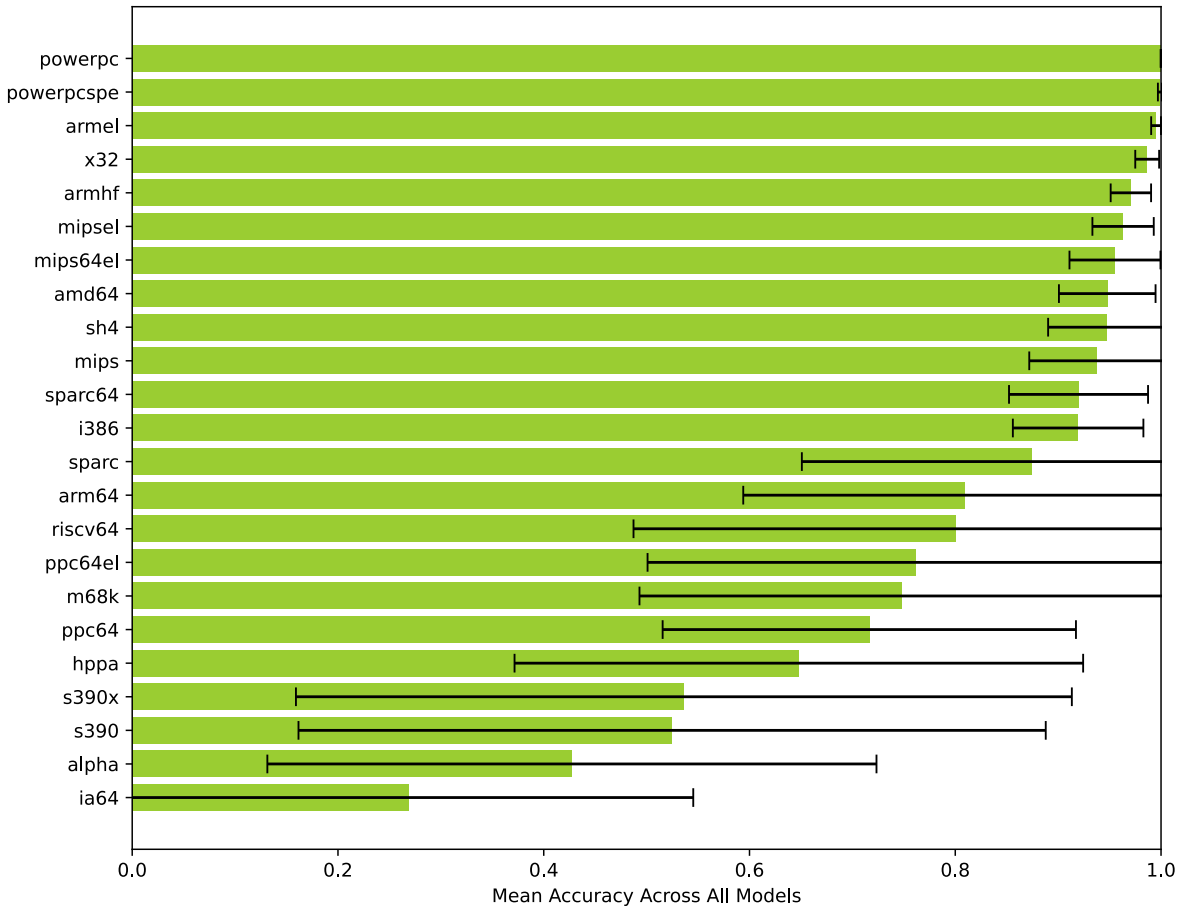


Figure 4.16: Instruction width type classification performance by ISA when using LOGO CV on the ISAdetect dataset. The error bars indicate the standard deviation across runs.

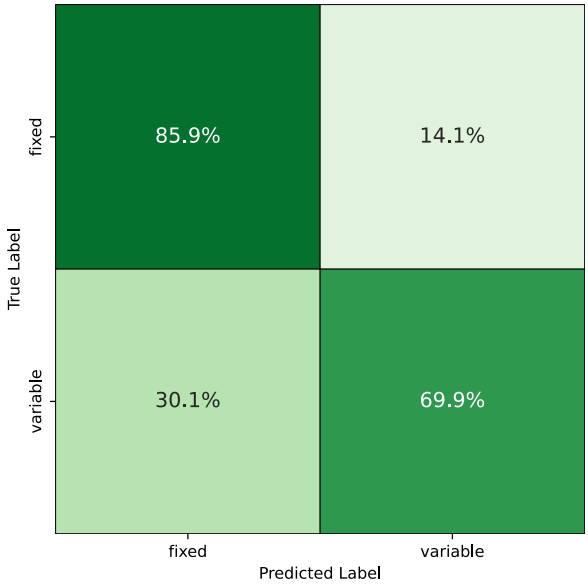


Figure 4.17: Confusion matrix of instruction width type classification when using LOGO CV on the ISAdetect dataset, aggregated across all models

Figure 4.17 aggregates the results across ISAs and models, and shows the confusion matrix. We see that the models are slightly biased towards predicting fixed instruction width.

4.2.3 Training on ISAdetect, testing on CpuRec

For the ISAdetect-CpuRec experiment, we train the models on the ISAdetect dataset and test them on the CpuRec dataset. Table 4.6 shows classification performance for every model/ISA combination.

Table 4.6: Instruction width type classification performance when training on the ISAdetect dataset and testing on the CpuRec dataset, across 20 runs for each model. Each row shows the number of correctly classified files over the total number of classification attempts. Overall mean accuracy and standard deviation across all runs are reported in bold at the bottom, along with a 95% confidence interval range around the mean.

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
6502	3/20	0/20	17/20	1/20	5/20	0/20
68HC08	4/20	3/20	8/20	7/20	5/20	1/20
68HC11	20/20	1/20	7/20	3/20	16/20	6/20
8051	4/20	0/20	20/20	0/20	16/20	1/20
Alpha	20/20	20/20	20/20	20/20	20/20	20/20
ARC32eb	0/20	6/20	12/20	4/20	2/20	2/20
ARC32el	8/20	11/20	20/20	7/20	5/20	13/20
ARcompact	4/20	2/20	5/20	11/20	1/20	2/20
ARM64	11/20	20/20	20/20	20/20	17/20	20/20
ARMeb	20/20	16/20	20/20	15/20	9/20	17/20
ARMel	20/20	20/20	20/20	20/20	20/20	20/20
ARMhf	20/20	20/20	19/20	20/20	20/20	20/20
AVR	4/20	1/20	0/20	1/20	5/20	1/20
AxisCris	13/20	1/20	18/20	2/20	11/20	0/20
Blackfin	6/20	1/20	0/20	3/20	2/20	1/20
CLIPPER	5/20	1/20	1/20	1/20	0/20	4/20
CompactRISC	3/20	2/20	5/20	3/20	3/20	1/20
Cray	0/20	4/20	0/20	8/20	0/20	2/20
Epiphany	0/20	0/20	0/20	0/20	0/20	0/20
FR-V	20/20	19/20	20/20	20/20	20/20	20/20
FR30	20/20	19/20	5/20	19/20	16/20	18/20
FT32	15/20	11/20	20/20	15/20	20/20	18/20
H8-300	11/20	4/20	19/20	7/20	4/20	1/20
H8S	0/20	0/20	11/20	0/20	2/20	0/20
HP-Focus	0/20	8/20	2/20	5/20	4/20	10/20
HP-PA	20/20	20/20	20/20	20/20	20/20	20/20
i860	5/20	7/20	11/20	3/20	14/20	11/20
IA-64	5/20	12/20	18/20	15/20	2/20	8/20
IQ2000	20/20	20/20	18/20	20/20	20/20	20/20
M32C	0/20	1/20	20/20	3/20	14/20	1/20
M32R	17/20	17/20	17/20	18/20	2/20	20/20
M68k	0/20	20/20	0/20	19/20	1/20	12/20
M88k	20/20	17/20	20/20	20/20	20/20	18/20
MCore	19/20	20/20	20/20	20/20	20/20	19/20
Mico32	20/20	20/20	20/20	20/20	20/20	20/20
MicroBlaze	20/20	20/20	20/20	20/20	19/20	20/20
MIPS16	19/20	17/20	10/20	11/20	19/20	18/20
MIPSeb	20/20	20/20	20/20	20/20	20/20	20/20

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
MIPSel	20/20	20/20	20/20	20/20	20/20	20/20
MMIX	20/20	12/20	20/20	17/20	20/20	19/20
MN10300	11/20	5/20	20/20	5/20	20/20	9/20
Moxie	0/20	5/20	6/20	5/20	1/20	5/20
MSP430	0/20	2/20	0/20	3/20	0/20	0/20
NDS32	4/20	12/20	9/20	18/20	5/20	0/20
NIOS-II	20/20	20/20	20/20	19/20	20/20	19/20
PDP-11	3/20	2/20	3/20	1/20	2/20	0/20
PIC10	11/20	12/20	5/20	15/20	20/20	20/20
PIC16	12/20	19/20	12/20	18/20	20/20	19/20
PIC18	18/20	14/20	0/20	13/20	0/20	17/20
PIC24	17/20	20/20	17/20	20/20	20/20	19/20
PPCeb	20/20	20/20	20/20	20/20	20/20	20/20
PPCel	20/20	20/20	20/20	20/20	20/20	20/20
RISC-V	20/20	20/20	20/20	20/20	18/20	20/20
RL78	14/20	1/20	20/20	3/20	17/20	3/20
ROMP	1/20	5/20	4/20	12/20	0/20	0/20
RX	13/20	0/20	20/20	1/20	19/20	1/20
S-390	16/20	20/20	20/20	20/20	20/20	19/20
SPARC	20/20	20/20	1/20	20/20	11/20	20/20
STM8	0/20	0/20	0/20	0/20	0/20	0/20
Stormy16	2/20	1/20	0/20	1/20	0/20	2/20
SuperH	20/20	20/20	17/20	20/20	20/20	20/20
TILEPro	0/20	0/20	15/20	0/20	6/20	3/20
TLCS-90	1/20	0/20	1/20	1/20	16/20	3/20
TMS320C2x	2/20	9/20	2/20	9/20	0/20	5/20
TMS320C6x	20/20	20/20	20/20	19/20	20/20	18/20
V850	0/20	5/20	0/20	2/20	1/20	0/20
VAX	0/20	18/20	0/20	20/20	7/20	9/20
Visium	20/20	5/20	20/20	4/20	20/20	14/20
X86	20/20	20/20	20/20	20/20	20/20	20/20
X86-64	16/20	20/20	19/20	20/20	9/20	20/20
Xtensa	0/20	5/20	0/20	5/20	0/20	4/20
XtensaEB	0/20	0/20	0/20	2/20	0/20	0/20
Z80	6/20	3/20	4/20	4/20	20/20	2/20
Overall	**0.536±0.064**	**0.532±0.037**	**0.601±0.037**	**0.560±0.039**	**0.566±0.040**	**0.531±0.033**
95% CI	**0.496–0.577**	**0.499–0.564**	**0.569–0.633**	**0.527–0.593**	**0.533–0.599**	**0.499–0.562**

Figure 4.18 aggregates the results across ISAs. None of the models perform any better than a baseline model with random output for this target feature.

Figure 4.19 aggregates results across the different models. While some architectures are mostly classified correctly, overall accuracy is low and very sensitive to random initialization.

Figure 4.20 aggregates the results across ISAs and models, and shows the confusion matrix. We see that the models are severely biased towards predicting fixed instruction width, to the point where true variable instruction width binaries are misclassified over 70.7% of the time.

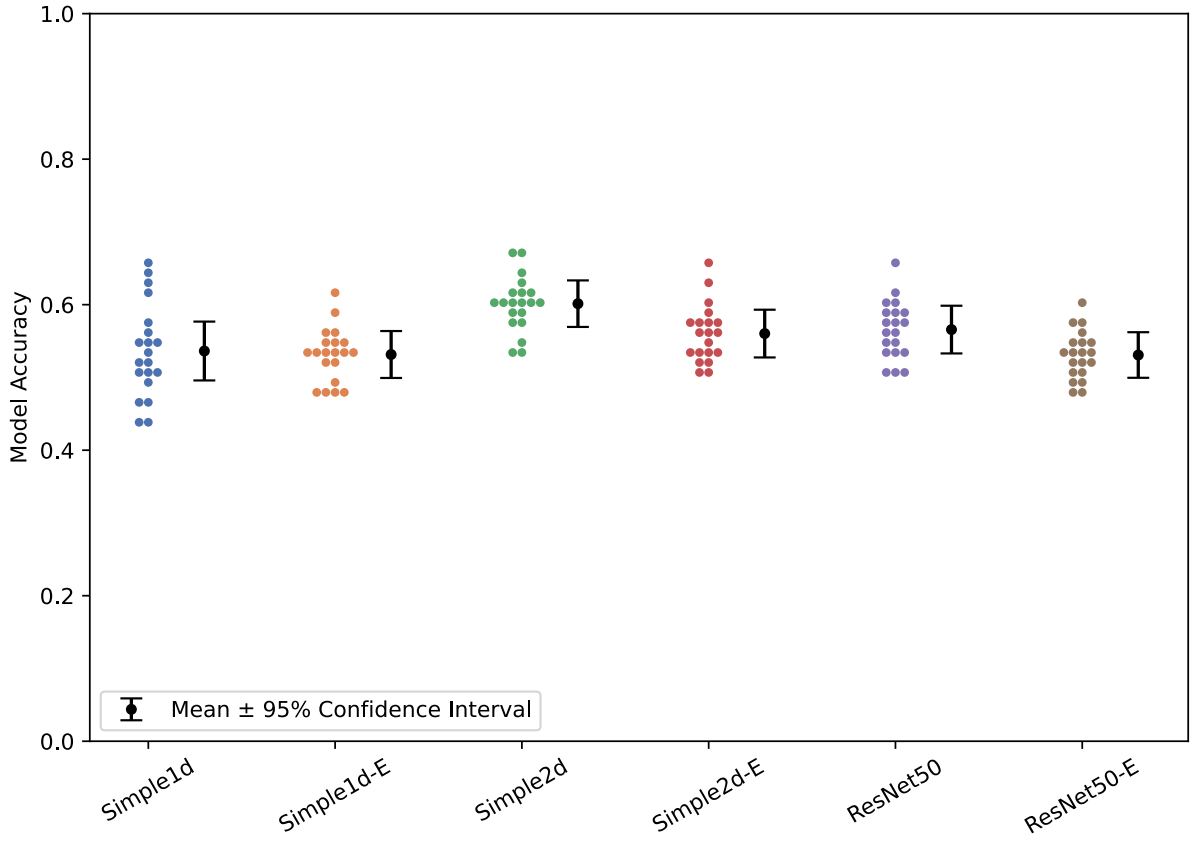


Figure 4.18: Instruction width type classification performance by model when training on the ISAdetect dataset and testing on the CpuRec dataset. The data points in the swarm plot represents the model's overall accuracy for each of the 20 runs.

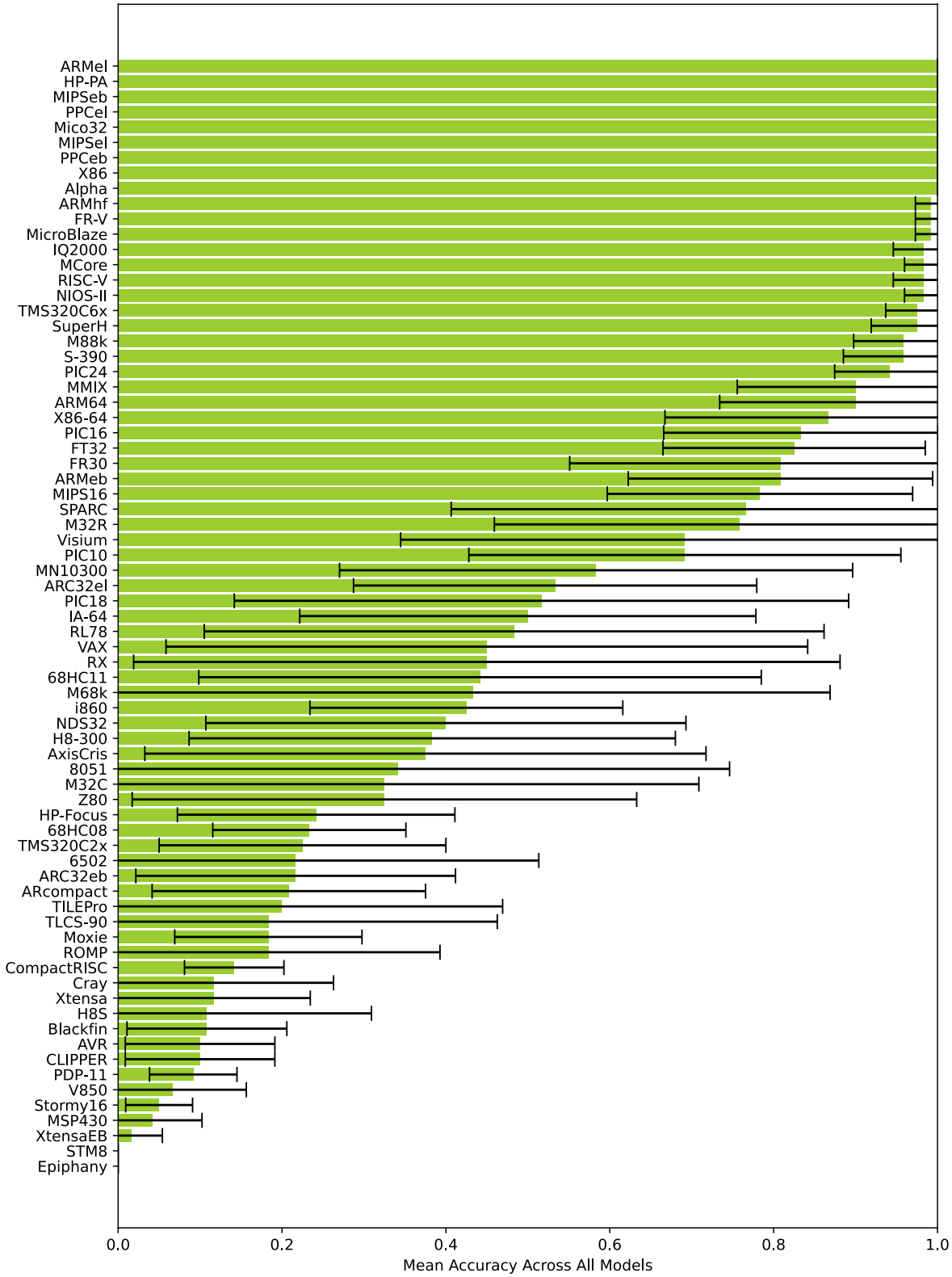


Figure 4.19: Instruction width type classification performance by ISA when training on the ISAdetect dataset and testing on the CpuRec dataset. The error bars indicate the standard deviation across runs.

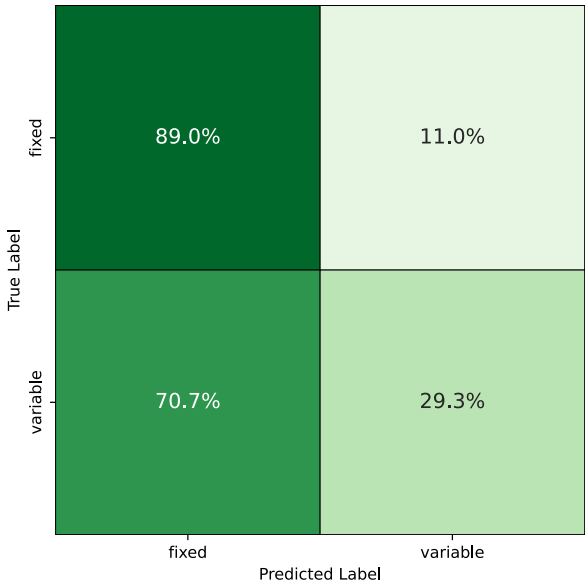


Figure 4.20: Confusion matrix of instruction width type classification when training on the ISAdetect dataset and testing on the CpuRec dataset, aggregated across all models

4.2.4 Training on ISAdetect, testing on BuildCross

For the ISAdetect-BuildCross experiment, each model is trained on the ISAdetect dataset and then performance-tested using the BuildCross dataset. Table 4.7 shows classification performance for every model/ISA combination.

Table 4.7: Instruction width type classification performance when training on the ISAdetect dataset and testing on the BuildCross dataset, across 20 runs. The reported performance numbers for each architecture are mean accuracy ± standard deviation. Overall mean accuracy and standard deviation across all runs are reported in bold at the bottom, along with a 95% confidence interval range around the mean.

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
arc	0.525±0.243	0.027±0.018	0.645±0.191	0.047±0.047	0.502±0.098	0.031±0.053
arceb	0.347±0.217	0.101±0.142	0.436±0.181	0.245±0.223	0.339±0.089	0.035±0.033
bfin	0.348±0.175	0.122±0.115	0.657±0.145	0.139±0.132	0.363±0.057	0.083±0.057
bpf	1.000±0.000	0.782±0.251	0.926±0.092	0.713±0.256	0.682±0.222	0.521±0.321
c6x	0.991±0.012	0.999±0.002	0.999±0.001	0.995±0.019	0.997±0.002	0.898±0.165
cr16	0.159±0.134	0.037±0.071	0.091±0.066	0.068±0.105	0.017±0.009	0.126±0.187
cris	0.596±0.228	0.322±0.313	0.779±0.175	0.382±0.296	0.762±0.081	0.095±0.098
csky	0.007±0.008	0.227±0.216	0.051±0.035	0.437±0.277	0.018±0.008	0.074±0.101
epiphany	0.080±0.074	0.011±0.017	0.180±0.107	0.024±0.022	0.065±0.028	0.010±0.014
fr30	0.209±0.171	0.273±0.353	0.866±0.107	0.245±0.325	0.474±0.124	0.076±0.116
frv	0.951±0.039	0.867±0.179	0.968±0.014	0.814±0.195	0.941±0.040	0.930±0.100
ft32	0.993±0.011	0.944±0.130	0.997±0.004	0.967±0.044	0.999±0.002	0.938±0.107
h8300	0.532±0.240	0.820±0.237	0.533±0.148	0.821±0.250	0.297±0.124	0.445±0.330
iq2000	0.958±0.059	0.966±0.049	0.993±0.009	0.947±0.037	0.988±0.015	0.773±0.270
kvx	0.187±0.126	0.352±0.263	0.098±0.067	0.372±0.162	0.104±0.031	0.105±0.136
lm32	0.999±0.001	0.896±0.145	0.869±0.076	0.973±0.043	0.929±0.042	0.946±0.078
loongarch64	0.909±0.090	0.704±0.232	0.553±0.114	0.733±0.212	0.743±0.129	0.902±0.116
m32r	0.962±0.024	0.768±0.251	0.963±0.017	0.937±0.057	0.826±0.077	0.879±0.152
m68k-elf	0.037±0.037	0.604±0.146	0.383±0.137	0.702±0.111	0.210±0.042	0.464±0.185

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
mcore	0.491±0.220	0.628±0.310	0.709±0.179	0.536±0.286	0.985±0.012	0.878±0.217
mcoreeb	0.398±0.215	0.821±0.213	0.605±0.221	0.807±0.212	0.978±0.019	0.878±0.182
microblaze	0.999±0.000	0.918±0.155	0.999±0.001	0.981±0.050	0.979±0.016	0.962±0.096
microblazeel	0.999±0.001	0.882±0.098	0.999±0.001	0.951±0.046	0.996±0.003	0.981±0.026
mmix	0.929±0.117	0.877±0.213	0.964±0.039	0.931±0.079	0.947±0.030	0.967±0.056
mn10300	0.720±0.190	0.037±0.036	0.842±0.121	0.053±0.034	0.979±0.037	0.038±0.050
moxie	0.048±0.048	0.388±0.203	0.116±0.108	0.444±0.163	0.075±0.050	0.364±0.212
moxieel	0.048±0.038	0.210±0.174	0.149±0.116	0.360±0.217	0.085±0.049	0.175±0.131
msp430	0.042±0.043	0.011±0.016	0.026±0.034	0.071±0.139	0.002±0.003	0.038±0.066
nds32	0.181±0.113	0.271±0.155	0.619±0.111	0.494±0.171	0.431±0.093	0.096±0.076
nios2	0.858±0.108	0.958±0.065	0.920±0.061	0.950±0.085	0.712±0.102	0.979±0.046
or1k	0.927±0.062	0.628±0.315	0.438±0.106	0.839±0.159	0.459±0.086	0.775±0.192
pru	0.973±0.024	0.945±0.053	0.985±0.008	0.872±0.248	0.277±0.143	0.978±0.062
rl78	0.605±0.215	0.189±0.202	0.938±0.057	0.190±0.244	0.981±0.015	0.163±0.179
rx	0.362±0.216	0.158±0.188	0.732±0.164	0.200±0.164	0.717±0.077	0.086±0.152
tilegx	0.645±0.226	0.601±0.389	0.969±0.025	0.769±0.338	0.990±0.005	0.665±0.332
tricore	0.433±0.226	0.154±0.233	0.701±0.191	0.315±0.264	0.813±0.106	0.284±0.254
v850	0.022±0.018	0.187±0.269	0.055±0.045	0.140±0.184	0.003±0.002	0.036±0.056
visium	0.973±0.031	0.762±0.173	0.995±0.006	0.765±0.221	0.987±0.007	0.804±0.181
xstormy16	0.466±0.202	0.622±0.302	0.851±0.064	0.614±0.274	0.384±0.099	0.193±0.180
xtensa	0.451±0.234	0.107±0.124	0.467±0.199	0.153±0.105	0.105±0.042	0.130±0.117
Overall	**0.634±0.032**	**0.585±0.054**	**0.696±0.025**	**0.648±0.058**	**0.640±0.011**	**0.567±0.048**
95% CI	**0.619–0.650**	**0.560–0.611**	**0.685–0.708**	**0.621–0.675**	**0.635–0.646**	**0.545–0.590**

Figure 4.21 aggregates the results across ISAs. With instruction width type as the target feature, it is even more prominent that the embedding models do not perform well when evaluating on the BuildCross dataset. The best-performing model is the *Simple2d* model, with a mean accuracy of 69.6%.

Figure 4.22 aggregates results across the different models. Once again, certain ISAs are consistently performing well, while others are systematically misclassified.

Figure 4.23 aggregates the results across ISAs and models, and shows the confusion matrix. We see that the models are severely biased towards predicting fixed instruction width, to the point where true variable instruction width binaries are misclassified 68.7% of the time.

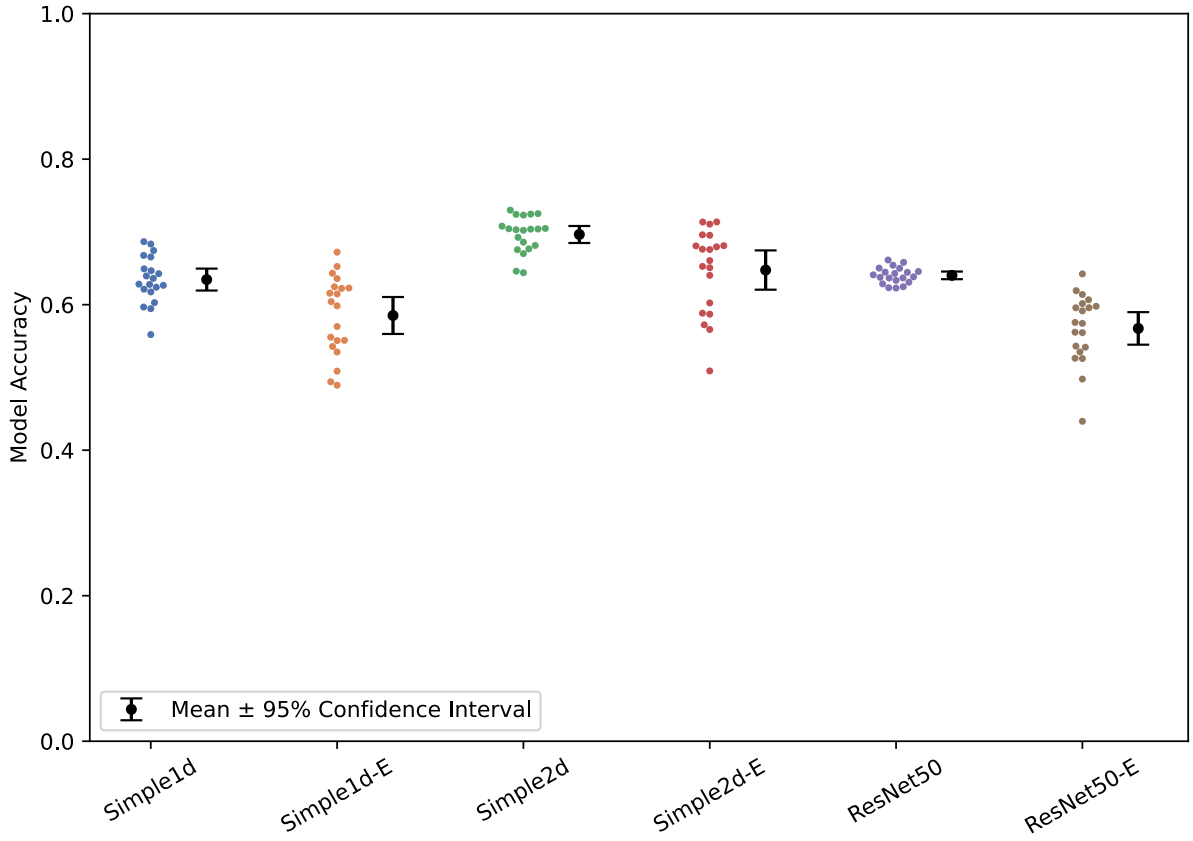


Figure 4.21: Instruction width type classification performance by model when training on the ISAdetect dataset and testing on the BuildCross dataset. The data points in the swarm plot represents the model's overall accuracy for each of the 20 runs.

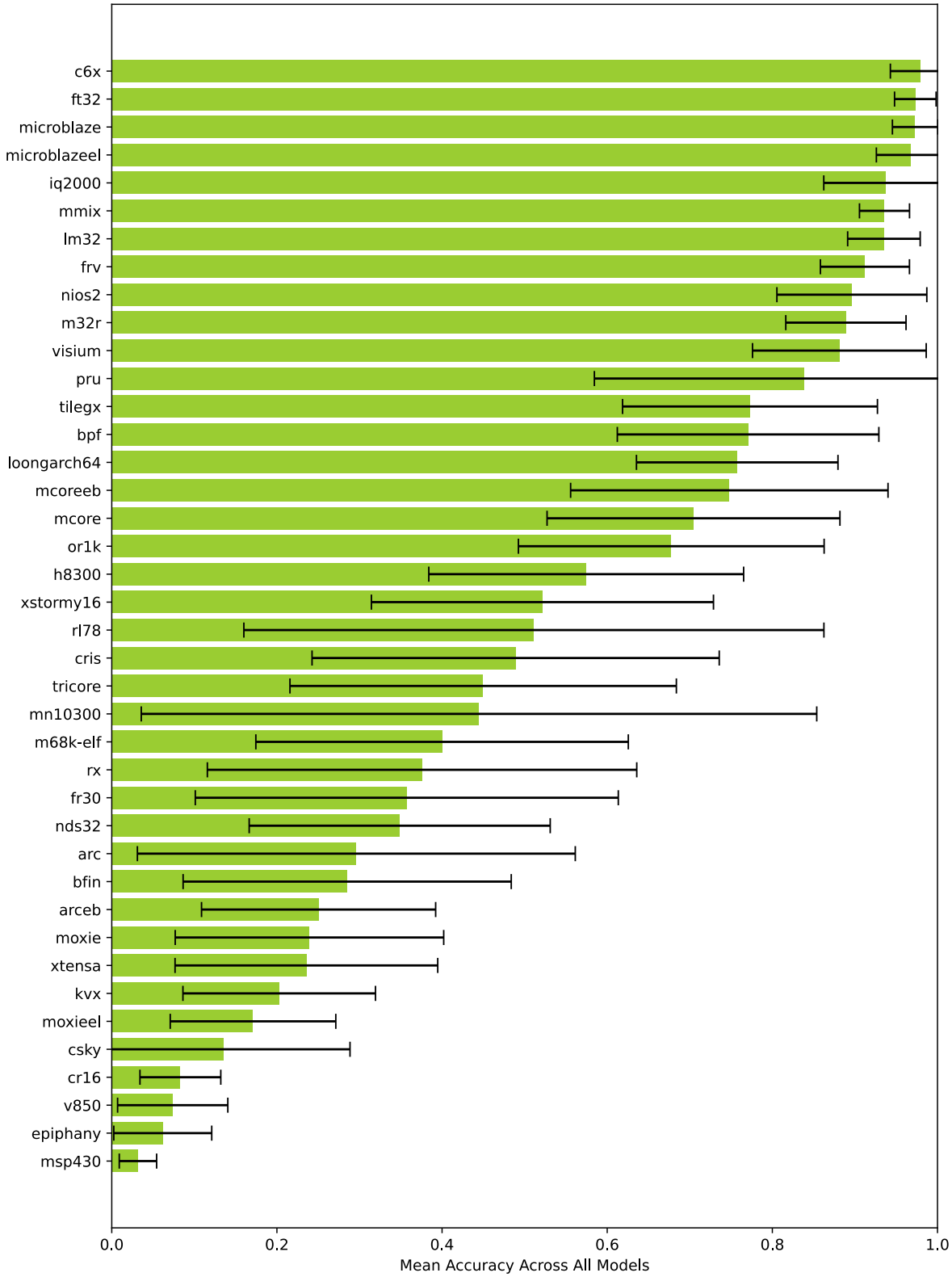


Figure 4.22: Instruction width type classification performance by ISA when training on the ISAdetect dataset and testing on the BuildCross dataset. The error bars indicate the standard deviation across runs.

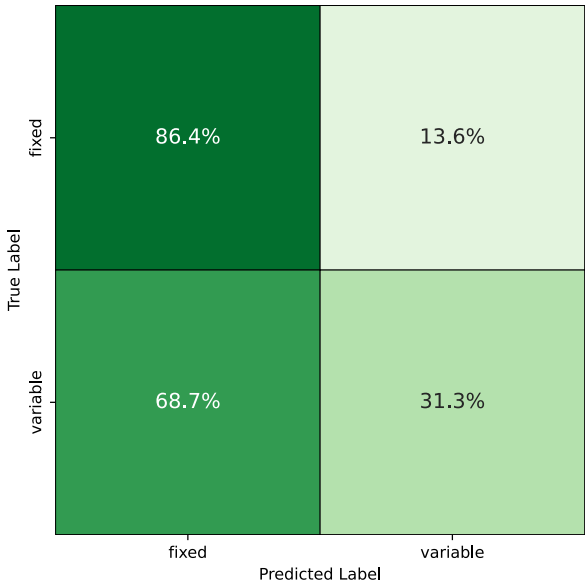


Figure 4.23: Confusion matrix of instruction width type classification when training on the ISAdetect dataset and testing on the BuildCross dataset, aggregated across all models

4.2.5 Training on ISAdetect and BuildCross, testing on CpuRec

For the Combined-CpuRec experiment, we train the models on the ISAdetect and BuildCross datasets, and test them on the CpuRec dataset. Table 4.8 shows classification performance for every model/ISA combination.

Table 4.8: Instruction width type classification performance when training on the ISAdetect and BuildCross datasets, and testing on the CpuRec dataset, across 20 runs for each model. Each row shows the number of correctly classified files over the total number of classification attempts. Overall mean accuracy and standard deviation across all runs are reported in bold at the bottom, along with a 95% confidence interval range around the mean.

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
6502	20/20	3/20	19/20	8/20	20/20	15/20
68HC08	20/20	18/20	18/20	20/20	20/20	19/20
68HC11	20/20	20/20	19/20	20/20	20/20	17/20
8051	20/20	20/20	19/20	20/20	20/20	20/20
Alpha	20/20	20/20	20/20	20/20	19/20	20/20
ARC32eb	1/20	19/20	0/20	20/20	0/20	14/20
ARC32el	7/20	18/20	0/20	20/20	3/20	16/20
ARcompact	6/20	20/20	19/20	20/20	14/20	20/20
ARM64	0/20	20/20	2/20	20/20	5/20	20/20
ARMeb	20/20	20/20	20/20	20/20	20/20	14/20
ARMel	20/20	20/20	20/20	18/20	20/20	19/20
ARMhf	19/20	20/20	19/20	16/20	7/20	16/20
AVR	15/20	10/20	6/20	9/20	18/20	16/20
AxisCris	20/20	19/20	19/20	20/20	20/20	19/20
Blackfin	20/20	19/20	19/20	20/20	20/20	18/20
CLIPPER	5/20	12/20	2/20	19/20	14/20	19/20
CompactRISC	20/20	20/20	19/20	20/20	18/20	19/20

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
Cray	20/20	13/20	19/20	19/20	19/20	17/20
Epiphany	1/20	18/20	10/20	20/20	12/20	20/20
FR-V	20/20	20/20	20/20	20/20	20/20	20/20
FR30	7/20	1/20	3/20	0/20	7/20	0/20
FT32	17/20	13/20	18/20	15/20	18/20	16/20
H8-300	20/20	20/20	19/20	20/20	18/20	20/20
H8S	20/20	17/20	18/20	19/20	16/20	19/20
HP-Focus	18/20	7/20	18/20	13/20	4/20	3/20
HP-PA	13/20	20/20	20/20	20/20	17/20	20/20
i860	0/20	4/20	1/20	2/20	4/20	10/20
IA-64	19/20	18/20	19/20	19/20	20/20	16/20
IQ2000	18/20	20/20	18/20	20/20	12/20	19/20
M32C	20/20	13/20	19/20	20/20	20/20	20/20
M32R	15/20	20/20	10/20	17/20	8/20	10/20
M68k	20/20	20/20	19/20	20/20	19/20	20/20
M88k	20/20	20/20	20/20	20/20	20/20	20/20
MCore	12/20	17/20	4/20	18/20	16/20	17/20
Mico32	20/20	16/20	20/20	20/20	20/20	17/20
MicroBlaze	20/20	20/20	20/20	20/20	20/20	20/20
MIPS16	0/20	1/20	1/20	0/20	4/20	0/20
MIPSeb	20/20	20/20	20/20	20/20	20/20	20/20
MIPSel	17/20	20/20	20/20	20/20	20/20	20/20
MMIX	20/20	20/20	20/20	20/20	20/20	20/20
MN10300	20/20	20/20	18/20	20/20	20/20	20/20
Moxie	20/20	20/20	19/20	20/20	20/20	20/20
MSP430	20/20	20/20	19/20	20/20	20/20	20/20
NDS32	20/20	20/20	19/20	20/20	15/20	20/20
NIOS-II	20/20	20/20	1/20	20/20	11/20	20/20
PDP-11	20/20	19/20	17/20	20/20	9/20	17/20
PIC10	0/20	2/20	2/20	0/20	1/20	0/20
PIC16	7/20	9/20	4/20	1/20	2/20	4/20
PIC18	0/20	1/20	1/20	0/20	0/20	0/20
PIC24	6/20	19/20	20/20	13/20	19/20	10/20
PPCeb	20/20	20/20	20/20	20/20	20/20	20/20
PPCel	20/20	20/20	20/20	20/20	20/20	20/20
RISC-V	20/20	18/20	20/20	20/20	20/20	20/20
RL78	20/20	20/20	19/20	20/20	20/20	20/20
ROMP	20/20	6/20	19/20	20/20	19/20	19/20
RX	20/20	20/20	19/20	20/20	20/20	20/20
S-390	20/20	19/20	9/20	20/20	10/20	20/20
SPARC	17/20	20/20	11/20	20/20	16/20	20/20
STM8	2/20	9/20	1/20	9/20	3/20	11/20
Stormy16	10/20	10/20	16/20	19/20	9/20	17/20
SuperH	10/20	19/20	13/20	19/20	5/20	11/20
TILEPro	5/20	10/20	18/20	18/20	19/20	15/20
TLCS-90	20/20	20/20	19/20	20/20	20/20	20/20
TMS320C2x	20/20	20/20	18/20	20/20	10/20	19/20
TMS320C6x	20/20	8/20	20/20	0/20	20/20	6/20
V850	20/20	14/20	19/20	20/20	20/20	18/20

Architecture	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
VAX	0/20	1/20	1/20	0/20	6/20	6/20
Visium	20/20	20/20	19/20	20/20	20/20	20/20
X86	20/20	20/20	19/20	20/20	20/20	20/20
X86-64	18/20	20/20	19/20	20/20	17/20	20/20
Xtensa	0/20	0/20	0/20	0/20	0/20	0/20
XtensaEB	0/20	20/20	15/20	20/20	7/20	20/20
Z80	20/20	20/20	19/20	20/20	20/20	19/20
Overall	**0.743±0.026**	**0.795±0.031**	**0.732±0.071**	**0.829±0.017**	**0.733±0.038**	**0.806±0.033**
95% CI	**0.716–0.770**	**0.768–0.821**	**0.691–0.773**	**0.807–0.852**	**0.703–0.763**	**0.780–0.833**

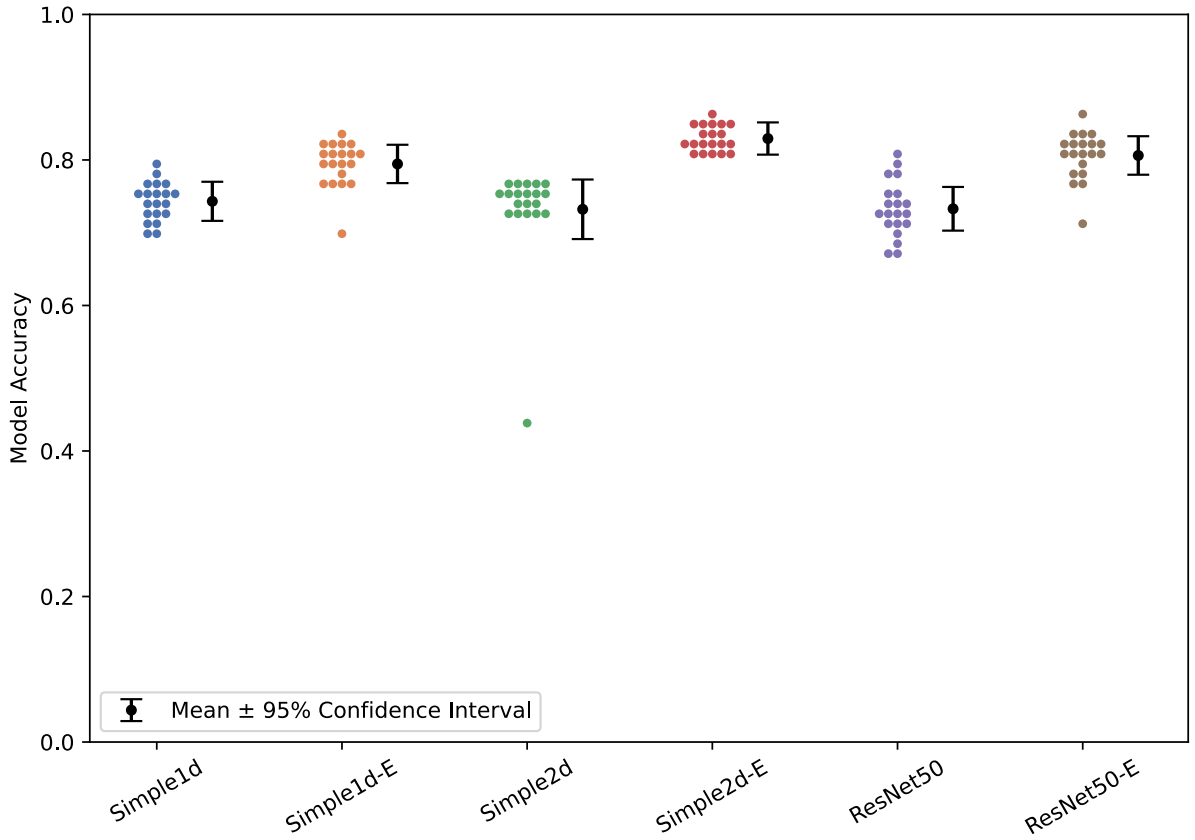


Figure 4.24: Instruction width type classification performance by model when training on the ISAdetect and BuildCross datasets, and testing on the CpuRec dataset. The data points in the swarm plot represents the model's overall accuracy for each of the 20 runs.

Figure 4.24 aggregates the results across ISAs, and Figure 4.25 aggregates results across the different models. For this target feature, we also see a significant performance improvement when adding the BuildCross dataset as additional training data. The *Simple2d-E* model performs the best, with a mean accuracy of 82.9%.

Figure 4.26 aggregates the results across ISAs and models, and shows the confusion matrix. Notably, adding the BuildCross dataset as additional training data seems to solve the bias issues observed in Figure 4.20, and the models are now reasonably balanced in their predictions.

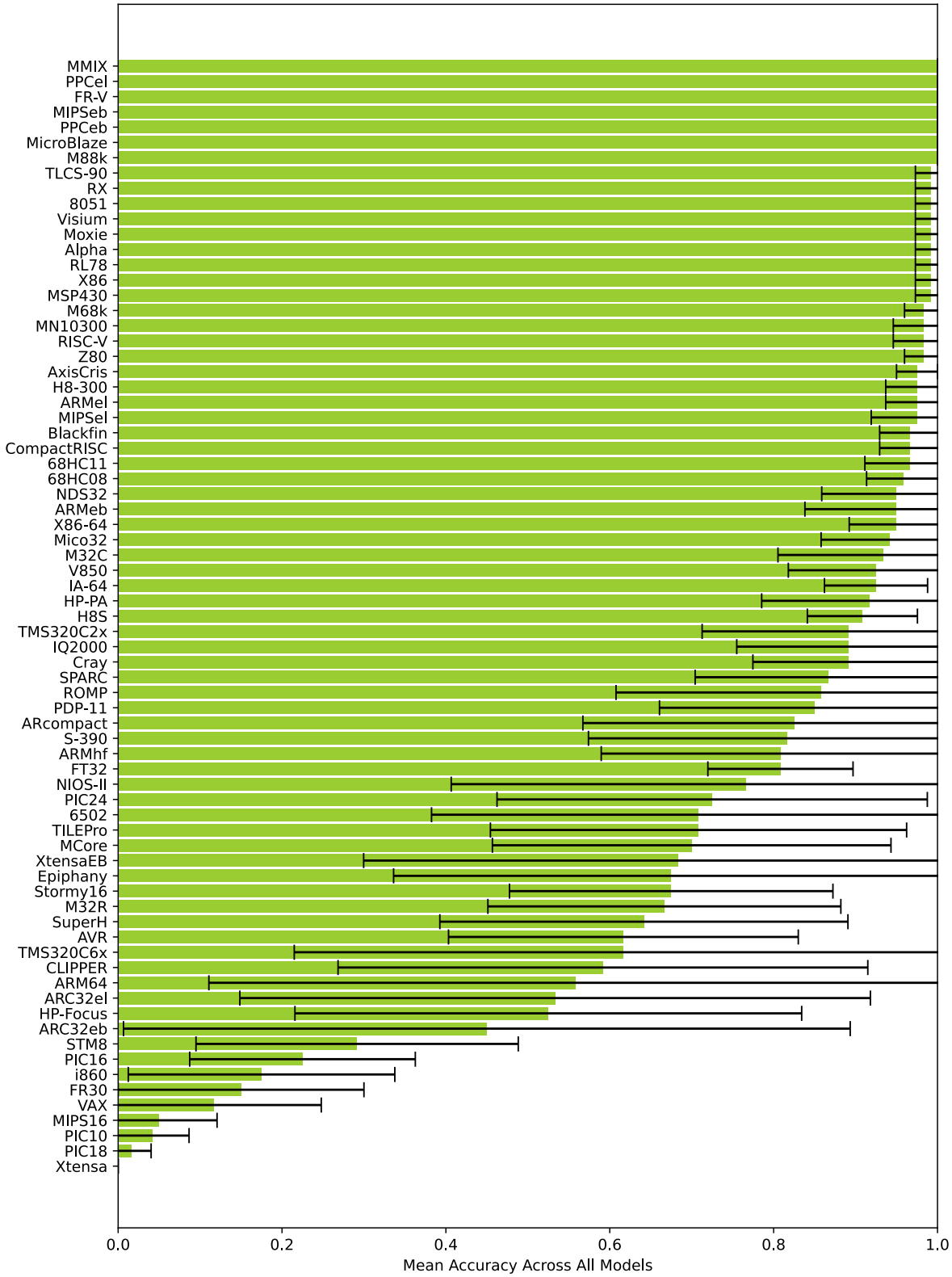


Figure 4.25: Instruction width type classification performance by ISA when training on the ISAdetect and BuildCross datasets, and testing on the CpuRec dataset. The error bars indicate the standard deviation across runs.

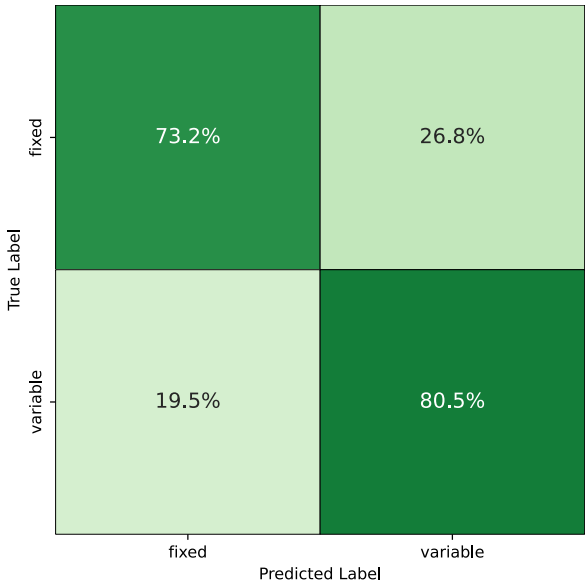


Figure 4.26: Confusion matrix of instruction width type classification when training on the ISAdetect and BuildCross datasets, and testing on the CpuRec dataset, aggregated across all models

Chapter 5

Discussion

This chapter discusses and interprets our results and findings from Chapter 4. In Section 5.1, we summarize the main discoveries and their implications. Section 5.2 then examines the performance characteristics of the different model architectures, focusing on aspects such as the impact of embedding layers, model complexity, CNN dimensionality, and observed performance variance. Then, Section 5.3 assesses the ability of our models to generalize to unseen ISAs. A comparative analysis with prior research is presented in Section 5.4. Following this, Section 5.5 critically evaluates the datasets employed, discussing their respective strengths and limitations. We also explore the broader sustainability implications in Section 5.6, and conclude by acknowledging the limitations of our work in Section 5.7.

5.1 Overview of key findings

In this section, we will briefly go over the main findings of our experiments. We will also highlight the potential implications of these findings related to our research questions by indicating further points of discussion for later sections in this chapter.

5.1.1 K-fold cross-validation and chosen evaluation strategies

In the initial K-fold cross-validation experiments on ISAdetect, we observed that all models performed similarly well, with near 100% classification accuracy on both endianness and instruction width type, as seen in Figure 4.1 and Figure 4.14 respectively. This result was also achieved with very little variance between runs, and points to the fact that for architectures previously seen during training, predicting endianness and instruction width is trivial. The near-perfect accuracy makes us suspicious that the models are quickly fitting to architectural features unrelated to endianness and instruction width. We believe we were correct in our assessment that this needed to be investigated further, and that Leave-One-Group-Out Cross-Validation (LOGO CV) as well as testing on different datasets is the best way to evaluate the general ability of Convolutional Neural Networks (CNNs) in detecting Instruction Set Architecture (ISA) features.

With LOGO CV and the evaluation strategies using multiple datasets, we observe more nuanced and varied results. In terms of raw classification performance, individual model performance varies considerably depending on the datasets used. With LOGO CV on ISAdetect, we achieve mean accuracies of **90.3%** for endianness detection (see Table 4.1) and **88.0%** for instruction width

type detection (see Table 4.5). Both of these scores were achieved with the Simple1d-E model. The performance on the other datasets is not as convincing. On the evaluation setups testing on CpuRec, we see greater variability between runs with lower accuracies, and lower overall accuracy on the BuildCross tests.

5.1.2 Endianness detection

For endianness detection, we see clear performance differences between embedding and non-embedding models. The embedding models perform better in three out of the four evaluation strategies, with a clear gap in performance under LOGO CV on ISAdetect. The only exception is ISAdetect-BuildCross, where the non-embedding versions of both the Simple1d and the Simple2d model outperform their embedding counterparts, with significantly lower variance in the accuracies as well (see Table 4.3 and Figure B.3). Even so, the large performance difference under LOGO CV on ISAdetect (see Table 4.1) and the gap in accuracy in the ISAdetect-CpuRec (see Table 4.2) and Combined-CpuRec experiments (see Table 4.4) indicate that the embedding models are better suited for endianness detection. This is backed by the paired t-test results in Figure B.2, Figure B.4, and Figure B.5, In these three evaluations, none of the non-embedding models outperform any of the embedding models with a p-value of less than 0.05.

5.1.3 Instruction width type detection

For instruction width type detection, it is less clear which model variations perform the best, especially compared to endianness detection. In the ISAdetect-CpuRec evaluation, the performance is poor, as seen in Table 4.6. The average accuracy measure ranges from **53.2%** to **60.1%** – results that are no better than always predicting the majority class. Just like with endianness detection, the two best-performing evaluations are LOGO CV on ISAdetect and Combined-CpuRec. We also observe that non-embedding models are seemingly better in the two worst-performing tests, ISAdetect-CpuRec and ISAdetect-BuildCross, as shown by the comparisons in Figure B.9 and Figure B.8. Based on LOGO CV on ISAdetect and the Combined-CpuRec evaluation, we initially concluded that embedding models perform better overall, although with smaller performance differences across the top-performing models, making the advantage less pronounced than for endianness detection.

However, we notice an interesting trend: the confusion matrices for the models only training on ISAdetect show that variable width architectures are often misclassified as fixed width architectures, while the opposite is not true. In the ISAdetect-CpuRec confusion matrix in Figure 4.20, we see that **70.7%** of variable width architectures are classified as fixed, a significant error across all models. We see a similar pattern with ISAdetect-BuildCross, shown in Figure 4.23. This suggests that instruction width type classification models trained on only ISAdetect do not transfer well to other unseen architectures.

5.1.4 Impact of BuildCross dataset

We believe that the addition of the BuildCross dataset has had a positive impact on the overall quality and robustness of our testing methodology, as shown in the results from both ISAdetect-BuildCross and Combined-CpuRec experiments. Our ISAdetect-BuildCross performance shows comparable results to ISAdetect-CpuRec on instruction width detection, validating our efforts in creating a dataset with more unconventional and diverse architectures. However, an interest-

ing anomaly appears with endianness detection, where the non-embedding models outperform their embedding counterparts with remarkably low variance, seen in Figure 4.8. This unexpected pattern contrasts the other experimental suites. There is some indication of similar behavior, although less pronounced, in the instruction width detection experiments with the same setup.

While testing on the BuildCross dataset showed some interesting patterns, the most significant impact in terms of overall accuracy appeared when combining ISAdetect and BuildCross for training the models. The addition of BuildCross significantly improved performance on the CpuRec evaluation, particularly for instruction width type detection. In this case, accuracy jumped from **50–60%** to above **80%** for the top performing models (comparing Table 4.6 and Table 4.8). This indicates that the addition of more diverse and unconventional architectures in the training data has a positive impact on the overall performance of our models. This is also further indicated by the improved balance of the models, as demonstrated in Figure 4.26. We find that variable-width architectures are no longer systematically misclassified as fixed, as was the case when training only on ISAdetect.

5.2 Model architecture performance analysis

5.2.1 Impact of embedding layers

In most of our experiments, we see that the model architectures that employ an embedding layer as the first layer of the model perform significantly better than their non-embedding counterparts. This is a key finding, and aligns with our hypothesis that embedding techniques may improve performance for CNN models due to the categorical nature of binary code. A possible reason for this can be demonstrated with the following example.

Consider this simple instruction for the Intel 8080 instruction set:

```
ADI 25;
```

It uses the `ADI` opcode, which indicates an addition with an immediate value. It sums the content of the accumulator register and the immediate value and saves the result to the accumulator register. We can examine what this looks like when assembled to a 16-bit binary instruction:

$$\underbrace{1100\ 0110}_{Opcode} \quad \underbrace{0001\ 1001}_{Immediate\ value}$$

The first byte contains the operation code. While operation codes are represented as numbers in the executable code, there is no semantic meaning to these numbers. It is a discrete, categorical piece of data that has no semantic relationship to bytes of close values such as $1100\ 0101$ and $1100\ 0111$.

Intuitively, an operation that is semantically similar to `ADI` (Add Immediate) is `SUI` (Sub Immediate). It performs the same operation but subtracts the immediate value from the accumulator instead of adding it. The opcode for `SUI` is $1101\ 0110$. Converting this to base 10, the numbers used to represent the `ADI` and `SUI` instructions are 198 and 214. These values themselves do not properly represent the close semantic relationship between the operations.

However, introducing an embedding layer in the model makes it capable of identifying and learning semantic relationships such as this by converting each byte value into a continuous vector. Bytes with close semantic relationships would be represented as similar vectors. While this is

a very simple example, converting categorical data into semantic-capturing vectors is a powerful technique that often results in superior performance when training and testing deep learning models on categorical input.

5.2.2 Model complexity

A clear trend in our results is that the larger ResNet models do not outperform the smaller and simpler CNN architectures; in many cases, they actually perform worse. This observation holds even when considering that ResNet models were given a larger input window of 1024 bytes compared to the 512 bytes used by simpler models. Figure 5.1 shows the large difference in parameter count between our models. A possible explanation for this is that the ResNet models' high representational power might overfit to the training data. This typically happens when the size or diversity of the training data is limited.

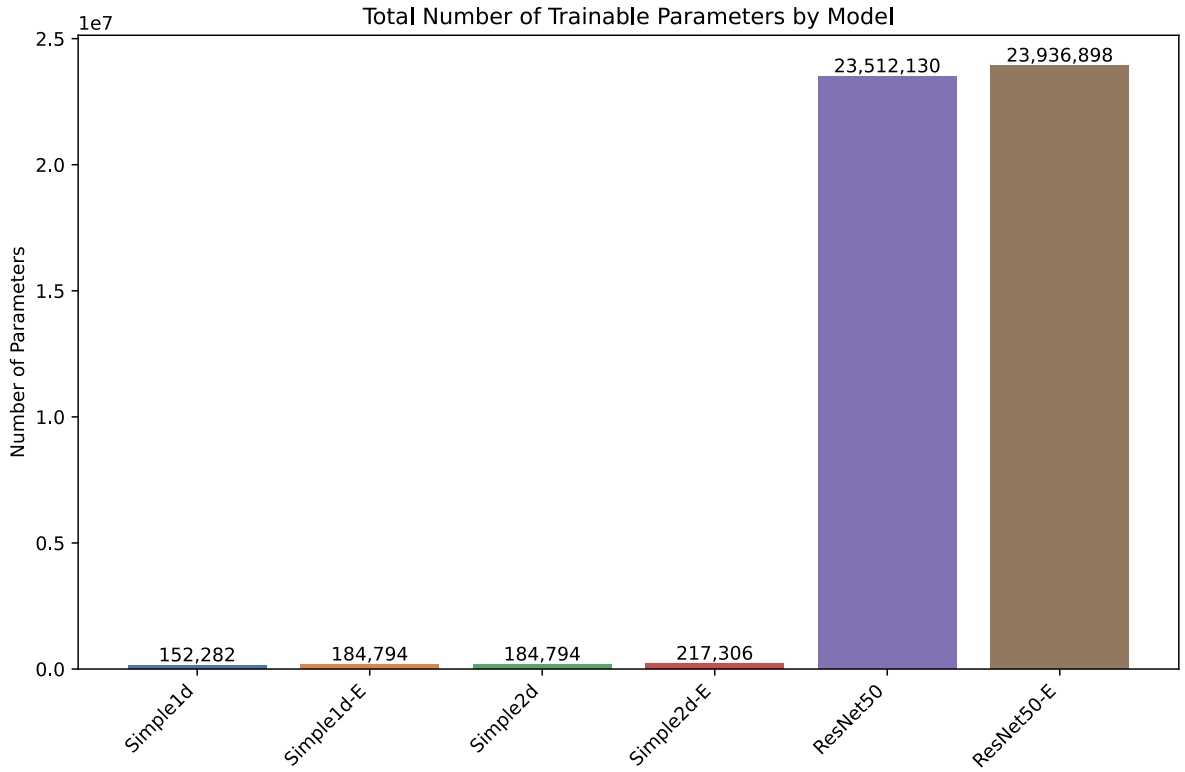


Figure 5.1: Parameter count of each model

While we consider our data quantity to be sufficient, there are reasons to believe that the diversity of the data is not high enough to avoid overfitting when training larger models. This claim is also supported by the fact that every model we trained converged rather quickly, almost always after just one or two epochs. In fact, the limited representational power of the smaller models may be beneficial in our case, since they are forced to learn simpler and more obvious patterns instead of picking up on what might effectively be random noise in the training data.

5.2.3 CNN dimensionality

While most applications of CNNs, such as image analysis, use two-dimensional convolution layers, we also included one-dimensional models in our experiments. Before running our

experiments, we hypothesized that two-dimensional CNNs might perform better than the one-dimensional ones due to the repeating patterns of fixed-width instruction sets. We also chose the 32x16 input size for the same reason, considering that many ISAs use 32-bit wide instructions.

Our results indicate that for detecting endianness, the two-dimensional models generally do not show an advantage over the one-dimensional counterparts. Likely, the models do not rely on repeating patterns for detecting endianness, since endianness fundamentally operates on an individual byte organization level.

For detecting instruction width type, the two-dimensional models do perform as well or better than the one-dimensional models for experiments that use CpuRec or BuildCross as the test set. However, for LOGO CV on ISAdetect, the one-dimensional models still perform slightly better.

The relationship between model dimensionality and performance appears to be influenced by both the specific architectural feature being detected and the diversity of the training/testing datasets. This indicates that optimal CNN dimensionality for binary code analysis may be feature-dependent, rather than universally favoring a particular approach.

5.2.4 Variance in model performance

When training a deep learning model, several components use pseudo-randomness:

- **Weight initialization:** The trainable model parameters are initialized with random values before training starts. This is usually preferred over starting with all parameters set to zero.
- **Mini-batch sampling:** For each training iteration, a random subset of the training data is used to compute the next weight update.
- **Dropout:** A random set of neurons in each layer is set to zero during training.

To control and reproduce these pseudo-random elements, one can specify a seed. Setting a seed guarantees that the pseudo-random behavior can be reproduced, and when developing our models, we train and test them multiple times using different seeds. This allows us to compare the accuracy between different random initializations.

Generally, we observe a very high variance between different runs due to differences in randomness. This indicates that the training process of the model is unstable, where the performance on an unseen test set varies substantially even if the training loss quickly converges to zero. For instance, the best-performing model for endianness detection (*Simple1d-E*), when evaluating with LOGO CV on ISAdetect, shows a standard deviation of up to 28 percentage points (p.p.) for certain ISAs when comparing the accuracy across different random seeds (see Table 4.1). This happens despite precautions such as using low learning rates and regularizing the models with dropout.

This is common behavior when the size of the training dataset is limited. While we consider our training dataset to be large and comprehensive, the model variability strengthens our suspicion that the dataset is too homogeneous for optimally training deep neural networks. Another factor that might cause these results is outliers in the data. Random initialization might make models more or less sensitive to outliers in the training data.

5.3 Model generalizability

A key objective of our models is to be able to generalize to ISAs that were not seen during training. This section analyzes the generalizability of our models and how our experiments support this objective.

5.3.1 Leave-one-group-out cross-validation

We use LOGO CV as our cross-validation method for the ISAdetect dataset. In contrast to standard K-fold cross-validation, LOGO CV tests how the model performs on a previously unseen group. This is a more realistic scenario for testing generalizability, since it simulates the real-world scenario where a model is deployed to a new ISA that was not seen during training.

To showcase the effectiveness of LOGO CV, we can compare the accuracy of our models when evaluated on LOGO CV to the accuracy when evaluated with standard K-fold cross-validation. As shown in Section 4.1.1, we observe extreme performance under 5-fold cross-validation, with all models achieving accuracies **above 99%**. In contrast, the same setup using LOGO CV gave an accuracy of **89.7%** for the best model, as observed in Section 4.2.2. It is clear that when the overall objective is to evaluate generalizability to unseen ISAs, testing on the same ISAs as the ones present in the training data results in artificially high performance.

5.3.2 Testing on other datasets

For evaluating the generalizability beyond the 23 ISAs present in the ISAdetect dataset, we use the CpuRec dataset as well as BuildCross, the custom dataset we developed for this thesis. These datasets provide a more diverse set of ISAs than the ISAdetect dataset. In particular, CpuRec contains binaries from 76 different ISAs, while BuildCross contains binaries from 40 different ISAs.

5.3.2.1 CpuRec

We observe that models trained on ISAdetect do not generalize well to the CpuRec dataset. While certain models appear to perform well, it is important to note that there is an overlap between the ISAs present in the ISAdetect and CpuRec datasets. Figure 5.2 illustrates this. Out of the 76 ISAs present in CpuRec, 16 of them are also present in ISAdetect.

This overlap of ISAs between the datasets is a limitation of our experiments, as it may lead to models memorizing specific ISAs characteristics rather than learning generalizable features. However, we can mitigate this by excluding the ISAs present in ISAdetect from the CpuRec dataset, and observe the performance using only the non-overlapping ISAs. Endianness classification performance after excluding the ISAs present in ISAdetect from the test set is shown in Figure 5.3. We observe that the model with the highest accuracy is now Simple2d, achieving an accuracy of **74.7%**, down from Simple1d-E's 81.0% when evaluated on the entire CpuRec dataset. For instruction width type classification, the effect of removing overlapping ISAs is even more pronounced. Figure 5.4 shows the instruction width classification performance after excluding the ISAs present in ISAdetect from the test set. Here, the best-performing model only achieves an accuracy of **44.9%**, which is worse than what a baseline model that always predicts the most common class would achieve.

We identify several potential reasons for the poor generalizability of our ISAdetect-trained models:

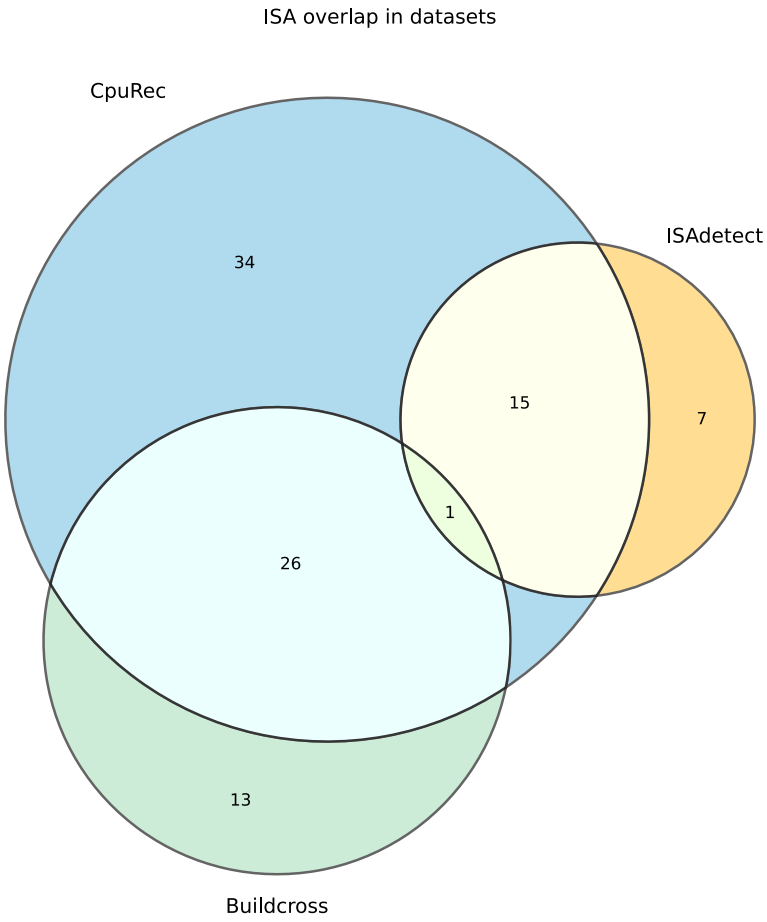


Figure 5.2: Venn diagram illustrating the overlap of ISAs present in the ISAdetect, CpuRec and BuildCross datasets

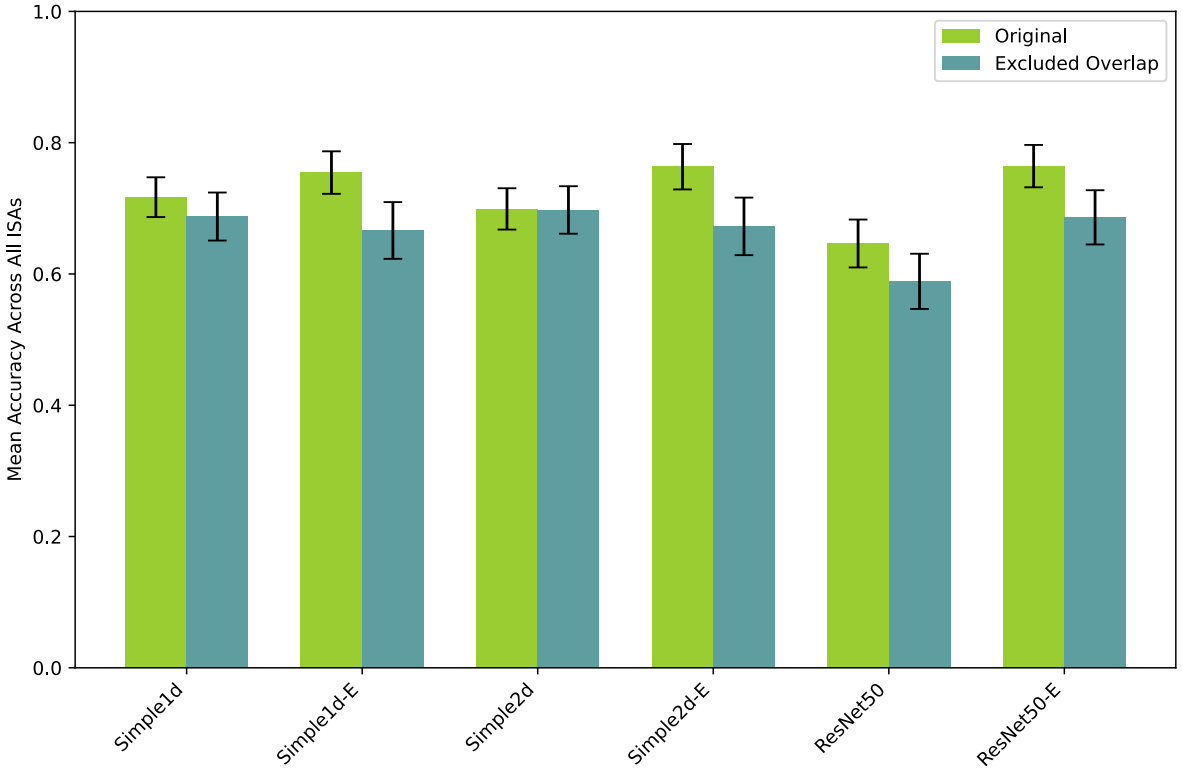


Figure 5.3: Endianness classification performance on the CpuRec dataset after excluding the ISAs present in ISAdetect

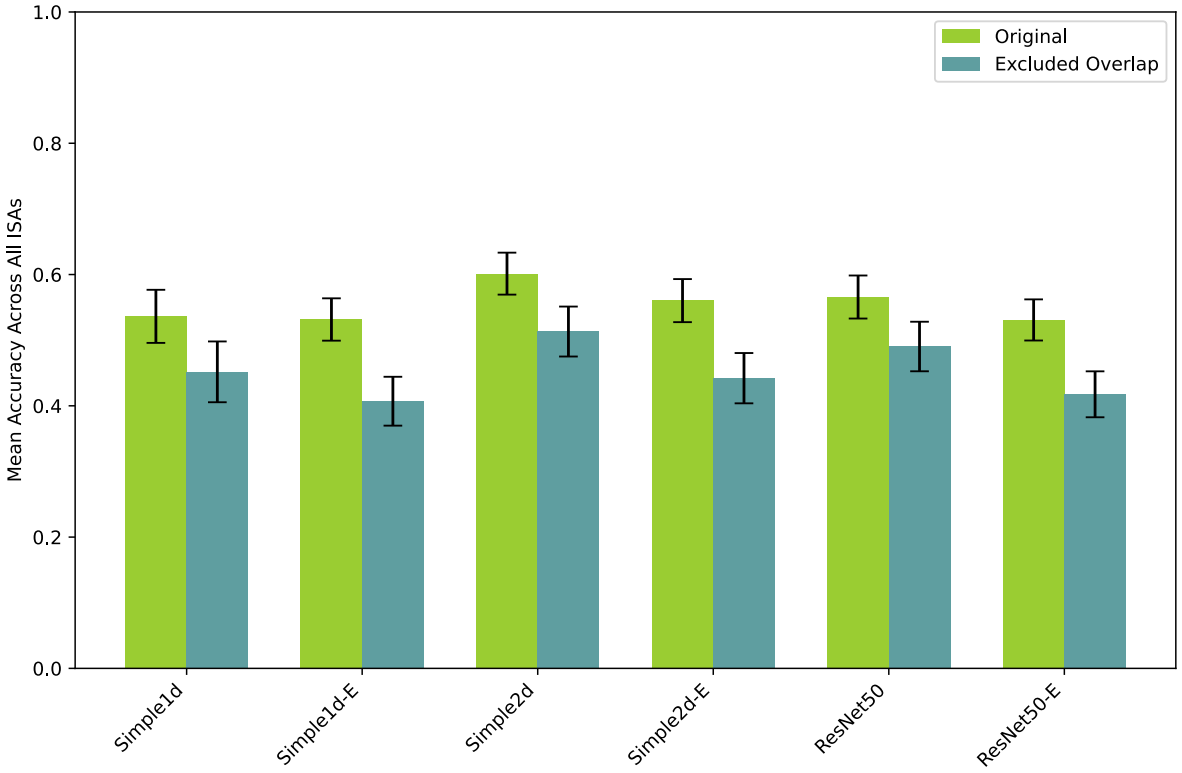


Figure 5.4: Instruction width classification performance on the CpuRec dataset after excluding the ISAs present in ISAdetect

- The diversity of the ISAdetect dataset used for training is somewhat limited. CpuRec contains 76 different ISAs, while ISAdetect only contains 23. In addition, the ISAdetect dataset is more homogeneous, with all ISAs being supported compile targets for recent versions of the Debian Linux distribution. CpuRec, on the other hand, was developed by curating binaries from a very diverse set of ISAs. By inspecting the ISA features in the two datasets, we can for instance observe that while all ISAs in ISAdetect have 32-bit or 64-bit word sizes, CpuRec also contains several ISAs with 8-bit and 16-bit word sizes which are likely very different in nature.
- The CpuRec dataset only contains a single binary file per ISA. This is a significant limitation of the dataset that makes our results less conclusive and more sensitive to anomalies in the specific binary used for each ISA.
- Due to the nature of deep learning, it is possible that the CNN models are picking up on ISA-specific patterns that are not inherently related to the endianness or instruction width. This is a common problem in deep learning and is known as overfitting to the training data. Since it is difficult to interpret the inner workings of CNN models, we can only speculate whether this is the case. However, the high accuracies observed when running K-fold cross-validation on the ISAdetect dataset do support the claim that the models are easy to fit to full ISAs compared to fitting them to specific ISA characteristics.

Our findings show that augmenting the training data with BuildCross does not considerably improve the generalizability of endianness detection. However, we do see indications that the instruction width type classification task benefits from augmenting the training data with BuildCross. To make this a fair comparison, we must note that training on BuildCross results in more ISAs that overlap between the training and test datasets, as compared to training on ISAdetect only. To emphasize that the performance is truly better on unseen ISAs, we can examine the results when excluding both the ISAs present in ISAdetect and BuildCross from the test set. Figure 5.5 illustrates this. Compared to Figure 5.4, we see significant performance improvements across all model architectures, indicating that the inclusion of a more diverse training dataset does improve the generalizability of instruction width type classification.

5.3.2.2 BuildCross

When evaluating on BuildCross, in contrast to the other experiments, we observe that the non-embedding models perform better. Particularly, the best-performing model for endianness classification is Simple1d, achieving an accuracy of **71.3%**. For instruction width type classification, the best-performing model is Simple2d, achieving an accuracy of **69.6%**.

An advantage of the BuildCross dataset compared to the CpuRec dataset is that there is little ISAs overlap with the training dataset (ISAdetect). This reduces the risk of artificially high performance numbers due to the models memorizing specific ISAs characteristics rather than learning generalizable features.

We note that while generalizability for the endianness classification task seems similar between the CpuRec and BuildCross datasets, the instruction width type classification task shows a clear improvement when evaluated on the BuildCross dataset.

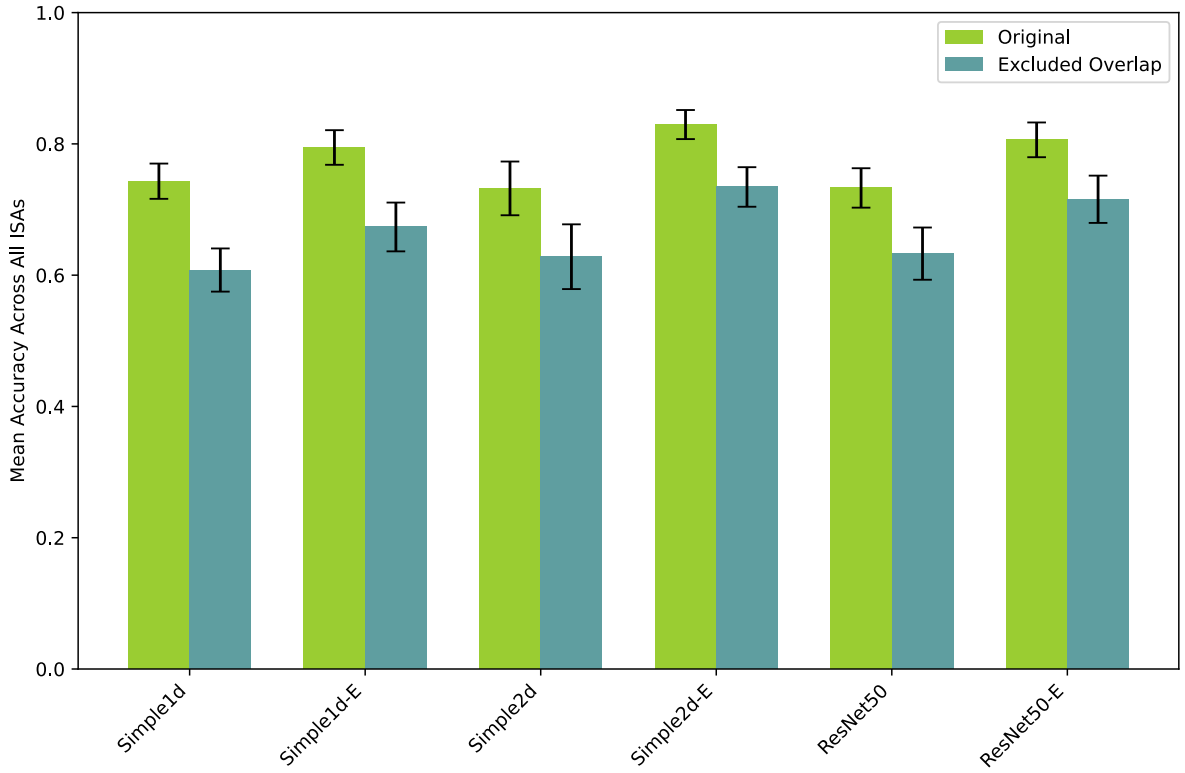


Figure 5.5: Instruction width classification performance on the CpuRec dataset after excluding the ISAs present in ISAdetect and BuildCross

5.4 Comparison with prior literature: Andreassen and Morrison

In our search for related work documented in Section 2.4, the thesis *Discovery of ISA features from binary programs from unknown instruction set architectures* by Andreassen and Morrison stands out as the only other identified research that specifically addresses the problem of detecting individual ISA features from unknown binary code [45]. We want to disclose that this work was supervised by Donn Morrison, who is also the supervisor of this thesis. For brevity in the following discussion, we will refer to this research as "Andreassen's work," acknowledging Morrison's supervisory role in the project. The thesis uses similar evaluation strategies and datasets, but with different feature extraction methods. Andreassen uses explicit feature engineering with classical machine learning classifiers for predicting the different ISA features, as opposed to deep learning techniques that automatically extract features from the binary code. In addition to targeting endianness and instruction width type detection, he includes a third target feature of detecting the instruction width of fixed-width architectures.

The thesis uses experiments similar to ours for endianness detection, with the same datasets and evaluation strategies. We will compare the results of our models with the results of Andreassen's models where applicable. However, there are notable differences in dataset labeling and the architectures used for training and testing, which complicate a fully accurate and direct comparison. In the rest of this section, we present our interpretation of a direct performance comparison on endianness and instruction width type classification, before discussing the key differences in our approaches and addressing the issues associated with these comparisons.

5.4.1 Endianness

5.4.1.1 Performance

There are two experimental suites targeting endianness set up in [45] that are comparable to our results. The first suite is LOGO CV on ISAdetect on code sections, where he achieved an accuracy of **92.0%** using a Random Forest classifier with bigram features, and **91.7%** using a Random Forest classifier with EndiannessSignature features. In comparison, we achieve an accuracy of **90.3%** using the Simple1d-E model on the same dataset. Andreassen's accuracy figures are slightly higher than ours on average, but still within the 95% confidence interval of ±2.0% for our top-performing model (see Table 4.1). The other comparable suite is training on ISAdetect code sections and testing on CpuRec. Andreassen achieved an accuracy of **86.3%** using a Random Forest and Logistic Regression classifier with bigram features [45]. We were only able to achieve an accuracy of **75.4%**, **76.3%** and **76.4%** using the Simple1d-E, Simple2d-E and ResNet50-E models respectively in our testing (see Table 4.2). This is a relatively large difference in performance.

5.4.1.2 Complexity

Andreassen uses two different feature engineering techniques: bigrams and EndiannessSignatures. Both methods have different advantages in complexity and training data requirements compared to our CNN approach. The bigram feature consists of counting up all combinations of two byte-pairs in each program, resulting in a histogram of $256 \cdot 256 = 65,536$ input features. The EndiannessSignature feature consists of the counts of only 4 bigrams: `0xfffe`, `0xfeff`, `0x0001`, and `0x0100`, and the idea was originally developed by Clemens (see Section 2.4.1.1) [45, 46].

The bigram feature was limited to 150 binaries per architecture for training due to memory constraints but is still able to perform well with a much smaller training dataset than what deep learning approaches would require. The 65,536-long feature vector does result in a large number of trainable parameters depending on the classifier. Assuming that Andreassen used scikit-learn's MLPClassifier with the default hidden layer size of 100, this model would have 6.56 million weights [100]. In terms of model complexity, this is more than our simple CNN models, which have 150k–217k parameters, but considerably less than the 23.5 million parameters of ResNet50. The EndiannessSignature feature, on the other hand, is very simple with an input feature vector of only 4 elements. It is able to achieve similar performance and often outperforms the bigram feature. Overall, Andreassen's approaches are achieving similar performance compared to ours, yet with less training data and lower complexity, especially when considering the EndiannessSignature feature [45].

5.4.1.3 Generalizability

A possible explanation for the effectiveness of Andreassen's feature extraction methods is his use of entire binary files, as the two bigram-based features can gather statistical information from the complete binary. In contrast, CNN-based automatic feature extraction is limited to the input window of the model, which we have set to 512 and 1024 bytes for Simple and ResNet models, respectively. While requiring more training data, the CNN approach might have an advantage in inference scenarios when only limited portions of a binary are available. We suspect that the bigram feature, based on a memory-intensive brute force approach, would require larger amounts of input binary data to be effective. For the EndiannessSignature feature, we cannot determine how frequently these specific bigrams appear within the first kilobytes of code, making it diffi-

cult to assess its effectiveness on smaller code sections. While our CNN approach might be better suited for smaller code samples, Andreassen's models can make accurate predictions using much simpler classification techniques [45].

5.4.2 Instruction width type

5.4.2.1 Performance

For instruction width type detection, there are no directly comparable evaluations between our work and Andreassen's. He uses CpuRec for training in all experiments, using LOGO CV on CpuRec, testing on ISAdetectFull, and testing on ISAdetectCode. The decision not to train on ISAdetect is justified by the lack of variety in his labeling, resulting in only 2 variable width architectures: `amd64` and `i386`, both from the x86 family. His results are commendable when training on the more limited CpuRec dataset, with the best-performing models reaching accuracies of **86.0%** with LOGO CV on CpuRec and **92.2%** accuracy when testing on ISAdetect [45]. While not directly comparable, our evaluation using LOGO CV on ISAdetect performs on par in terms of accuracy on ISAdetect (**88.0%** with Simple1d-E from Table 4.5). Meanwhile, our evaluation training on ISAdetect and testing on CpuRec performs significantly worse with only **60.1%** accuracy (see Table 4.6). Only the Combined evaluation results in comparable performance (**82.9%** with Simple2d-E from Table 4.8).

5.4.2.2 Complexity

Comparing model performance, training data, and model complexity, Andreassen presents three feature extraction methods: ByteDifferencePrimes, AutoCorrelation, and Fourier. Andreassen does not explicitly detail the feature vector lengths of his models, but reading his thesis we are able to create likely estimates. ByteDifferencePrimes keeps track of distances between identical byte values in the binary, managing the first 50 prime factors of distances for each byte value. This should result in a feature vector of $50 \cdot 256 = 12,800$ elements. The AutoCorrelation feature depends on the lag range parameter, computing autocorrelation for each value across that range. The optimal lag range depended on the classifier but varied between 32 and 1024. The Fourier feature converts the binary into a frequency spectrum, and the number of input features depends on the decided frequency range. Like with the AutoCorrelation feature, the optimal frequency range depended on the classifier but varied between 16 and 512, resulting in $2 \cdot FrequencyRange + 1$ features [45]. Both AutoCorrelation and Fourier result in models much smaller in complexity than our CNNs, with AutoCorrelation seemingly performing on par with or better than our models. ByteDifferencePrimes is significantly more complex, and the larger classifiers are comparable to our Simple CNN models in complexity.

5.4.2.3 Generalizability

Looking at Andreassen's results using AutoCorrelation and ByteDifferencePrimes, there is an apparent advantage in being able to train on the architecturally diverse CpuRec dataset. Comparing our results, we see that our models struggle when moving from ISAdetect to CpuRec, and that we required the more diverse BuildCross dataset to achieve similar performance. Andreassen's results indicate that his feature extraction methods can learn generalizable features even when trained on a smaller dataset, and that the feature engineering approach would be able to outperform our CNN models in this case [45].

5.4.3 Differences in approach and comparison issues

While we attempt to compare our work with Andreassen's, there are some key differences in our approaches that make a direct comparison difficult. These include dataset labeling and ISA inclusion differences, lack of documentation of model hyperparameters, and statistical evidence for comparing our approaches [45].

5.4.3.1 Dataset labeling and ISA inclusion differences

The most significant difference between our and Andreassen's comparative analysis lies in the dataset labeling. ISAdetect contains mostly mainstream architectures with readily available documentation, whereas CpuRec comprises code sections from numerous rare, less documented architectures. This presents considerable challenges for accurate labeling, something that we experienced during dataset preparation documented in Section 3.1.1. Andreassen's research relied on labeling provided by another researcher at the Norwegian University of Science and Technology (NTNU) which lacked information for many architectures, likely because of the time-consuming process of categorizing endianness and instruction widths. We conducted our own review of each of the architecture's labeling across both ISAdetect and CpuRec datasets, which enabled us to include additional ISAs for endianness and instruction width type analysis. During this process, we also identified incorrect labels for several architectures in CpuRec, such as endianness for `MCore` and instruction width type for `AxisCris`, `CompactRISC`, `M32R`, among others. A full list of new and corrected labels can be found in Table A.4 and Table A.5. Additionally, CpuRec has undergone updates since Andreassen completed his thesis, introducing architectures like `ARC32el` and `ARC32eb` that were unavailable during his research [45]. It should be noted that we are not entirely confident in the accuracy of our labeling. We discuss this further in Section 5.5.2. These labeling challenges combined with the differences in dataset composition limit our capacity to draw definitive conclusions comparing our findings to Andreassen's previous work.

5.4.3.2 Experimental setup and statistical evidence

In an attempt to compare model complexity and performance, we discovered a lack of documentation on how Andreassen's custom software implements these models, particularly regarding hyperparameter configuration. Reproducing his results fell out of the scope of this thesis, so we are not able to verify his work nor determine if reproduction is feasible. Many classifiers from the scikit-learn library that he uses have numerous configurable parameters, and while he optimized some hyperparameters on certain models using grid search, Andreassen only explicitly documents that the `random_state` parameter was set to 42 for reproducibility, `max_iter` set to 10,000 for convergence, and `n_jobs` set to -1 to enable multi-core processing. We can only assume that he used default values from scikit-learn for non-specified hyperparameters. On the evaluations where our work overlaps with his, meaningful statistical comparison is difficult since Andreassen appears to have done only a single run for each model configuration. While his use of LOGO CV provides some statistical robustness, models dependent on random initialization would benefit from multiple runs with different seeds to document performance stability. This was evident in our experiments, where we sometimes observed high variability and clear outliers in our results, like in Figure 4.24. Attempts at reproduction of his work would require making assumptions about the unspecified hyperparameters, further complicating verification of his findings [45].

5.4.3.3 Analysis of generalizability

In terms of model generalizability across unseen architectures, we find Andreassen's work lacks a more thorough discussion of this aspect. While he implements LOGO CV on ISAdetect for endianness and LOGO CV on CpuRec for instruction width type, which provides valuable insights into performance on unseen architectures, our research demonstrates that testing across different datasets with overlapping architectures significantly impacts model performance, as discussed in Section 3.4.3. This cross-dataset testing represents an important dimension of generalizability that Andreassen does not explore to a point where we can fairly compare our approaches. However, it is still our opinion that feature engineering provides much clearer insights into what the models are fitting to, with features like EndiannessSignatures and AutoCorrelation offering more interpretability than those learned by a CNN.

5.4.4 Summary

Comparing our deep learning approach with Andreassen's feature engineering methods reveals important insights about both methodologies. Classical machine learning classifiers with engineered features for detecting endianness and instruction width type demonstrate several advantages: they appear to require less training data, offer better interpretability, and can achieve comparable or even superior performance. However, differences in evaluation setups, dataset labeling and architecture inclusion as well as limited documentation of hyperparameters and cross-run variation make direct comparisons challenging. Our work contributes additional rigor through multiple runs and statistical analysis, while elaborating on generalizability through our methodology and newly created dataset. We have shown that the CNN approach may offer advantages compared to Andreassen's work, such as when working with limited binary sections or when expanding to ISA features without requiring the extensive domain knowledge needed for effective feature engineering. We believe both approaches have their merits, and that future work could benefit from leveraging both the interpretability and efficiency of engineered features and the automatic feature extraction capabilities of deep learning models.

5.5 Dataset quality assessment

5.5.1 ISAdetect dataset

ISAdetect is our main dataset for training machine learning models. It contains 23 different ISAs, with each architecture containing between 2800 and 6000 binary program samples [36, 46] per architecture. We train our models on the code sections of these binary program samples, excluding the code sections that are smaller than 1024 bytes. Even after this exclusion we are left with **16 GB** worth of samples, considerably more than the 21 MB and 120 MB in CpuRec and BuildCross, respectively. Full details of ISAdetect's data size and features used in the thesis can be found in Table A.2. We decided not to perform file splitting on the ISAdetect dataset to augment the amount of training data, as preliminary results revealed that this did not improve model performance or generalizability. Taking this into consideration, Table 5.1 shows the number of samples per ISA in ISAdetect. This averages to 4,086 samples per ISA.

Table 5.1: Number of samples per ISA in ISAdetect.

ISA	No. samples
s390	5,118
sparc	4,923
armhf	3,674
i386	4,484
arm64	3,518
armel	3,814
sh4	5,854
amd64	4,059
riscv64	4,285
mipsel	3,693
s390x	3,511
powerpc	3,618
mips	3,547
m68k	4,313
ppc64el	3,521
x32	4,059
hppa	4,830
powerpcspe	3,922
alpha	3,952
sparc64	3,205
mips64el	4,280
ppc64	2,822
ia64	4,983
Total	**93,985**

We can examine the number of samples for each class for both of our target features. Table 5.2 and Table 5.3 show the number of samples per class for endianness and instruction width type, respectively. We can see that for both target features, there are more than 30,000 training instances per class. This should be sufficient for training even the most complex CNN models.

Table 5.2: Number of samples per class for endianness in ISAdetect.

Endianness	No. samples	Percentage
little	54,176	57.64%
big	39,809	42.36%

Table 5.3: Number of samples per class for instruction width type in ISAdetect.

Instruction width type	No. samples	Percentage
fixed	63,458	67.52%
variable	30,527	32.48%

We also observe that there is some class imbalance in the dataset, particularly for instruction width type, where more than two-thirds of the binaries have fixed-width instructions. While this level of class imbalance is generally considered acceptable, it might cause the models to bias slightly towards the majority class [101, 102].

While the amount of data and the class balance are sufficient for training large deep learning models, it is not particularly diverse in terms of the ISAs present in the dataset. Since all architectures are supported compile targets for recent versions of the Debian Linux distribution, these ISAs are likely built for running on general-purpose operating systems rather than in specialized applications such as embedded systems. There are also ISA pairs in the dataset that are relatively similar, such as `armel`/`armhf` and `powerpc`/`powerpcspe`. This might cause slightly misleading performance numbers when running LOGO CV, as the models may overfit to the similarity between these ISAs.

On the note of similar architectures, the most pronounced problem with the ISAdetect dataset is when looking at the diversity of ISAs in the variable instruction width class. The ISAs in this class are `amd64`, `i386`, `ia64`, `m68k`, `s390`, `s390x`, and `x32`. However, several of these architectures are closely related variants rather than truly distinct instruction sets. Both `amd64` and `i386` belong to the x86 family and share the same fundamental instruction set, differing mostly in their 64-bit versus 32-bit implementations. Similarly, we discovered that the `x32` architecture is an extension of `amd64`, allowing 32-bit applications with 32-bit pointer addressing to use the more modern `x86-64` platform [103]. The `s390` and `s390x` architectures follow the same pattern, with `s390x` being an extension of the `s390` architecture [104]. As noted in Section 5.1.3, we noticed a pattern of variable-width architectures being misclassified as fixed width in models trained on ISAdetect. We suspect this occurs because models are learning to overfit to the fixed width class due to the dataset's imbalance toward fixed width ISAs (**67.52%** of samples) and the limited true diversity within the variable width class. We suspect that rather than learning the fundamental characteristics of distinguishing variable-width from fixed-width instruction sets, models appear to be identifying superficial features common among the similar variable-width architecture families. If this is the case, then this is a significant limitation of the ISAdetect dataset for our purposes of instruction width detection, as it does not provide sufficient diversity in the variable width class to train models that can generalize well to unseen architectures.

5.5.2 CpuRec dataset

We use the CpuRec dataset for testing the performance and evaluating the generalizability of our trained models. Contrary to ISAdetect, it contains binaries from 76 different ISAs, providing us with a significantly more diverse set of architectures for evaluating our models. A full list of ISAs as well as their sizes and labels can be found in Table A.3. CpuRec is very small compared to our other two datasets, with only **20.98 MB** of data, and it contains only a single compiled binary file per ISA. We consider this a significant limitation of this dataset. Not only does this mean that the data is too limited to be used for any training of deep learning models, there is also only one sample binary per ISA we can use for evaluating whether the model generalizes to new ISAs. To claim stronger statistical significance of the per-ISA results, we would ideally need more than one sample for each. Even so, using the ISAdetect-CpuRec and Combined-CpuRec evaluations, we were able to confirm our hypothesis that architectural diversity and the presence of rare ISAs in the training data are important factors for the generalizability of our models, justifying its inclusion as a testing dataset.

In terms of dataset balance, the CpuRec dataset maintains a relatively even distribution across both target features. Table 5.4 and Table 5.5 present the sample counts for each class within these respective features. While endianness shows a slightly larger class imbalance, we consider this acceptable given that CpuRec serves exclusively as a testing dataset for our models. The limited number of binary files initially prompted us to explore file splitting to augment our testing data. However, our investigation revealed a critical issue with this approach. To meet binary size requirements for some of the more obscure architectures, the CpuRec developer has duplicated code sections within the binary files themselves. This duplication means that splitting these files would not generate genuinely independent samples, but instead create redundant data points containing identical code patterns. File splitting would not provide meaningful data augmentation on some of the ISAs, and we decided against implementing this strategy on all the architectures.

Table 5.4: Number of samples per class for endianness in CpuRec.

Endianness	No. samples	Percentage
big	23	41.07%
little	33	58.93%

Table 5.5: Number of samples per class for instruction width type in CpuRec.

Instruction Width Type	No. samples	Percentage
variable	41	56.16%
fixed	32	43.84%

An important limitation of the CpuRec dataset we want to highlight is its inconsistencies in labeling, data sourcing, and reproducibility. While the CpuRec repository is open source, the origin of the compiled binaries is, in our opinion, not sufficiently documented. Examining the code comments in the `cpu_rec.py` source code and the additional documentation in `cpu_rec_sstic_english.md` [71] reveals that the binaries came from a combination of precompiled binaries available online and the author's cross-compiler suite, but this documentation is incomplete [72]. Given these reproducibility challenges, we developed our own labeling process to properly categorize the endianness and instruction width of the ISAs. We began by building on previous work, specifically using the labels from Andreassen [45], but as discussed in Section 5.4.3.1, we discovered numerous missing and incorrectly labeled ISAs. As described in Section 3.1.1.2, we expanded our approach by directly consulting architecture documentation found online, which yielded significantly more labeled architectures (see Table A.5). However, we acknowledge that without knowing the origins of many binaries and lacking comprehensive analysis tools, we cannot verify these labels with complete certainty. Furthermore, for architectures without standard file headers, the original author had to make educated guesses about where the code sections were located within the binary files before extracting them for inclusion in the dataset, introducing an additional layer of uncertainty about the accuracy of the extracted code segments. In addition, while we have no reason to distrust the author regarding the legitimacy of the ISA classifications, we cannot independently confirm that each binary matches its filename. We were able to validate some ISA labels using disassemblers from the BuildCross cross-compiler suite, which did provide additional confidence in our approach.

Ultimately, despite our incomplete confidence in the dataset's labeling accuracy, we believe the benefits of including this diverse architectural coverage in our work justify its use.

5.5.3 BuildCross dataset

The BuildCross dataset is a custom dataset we developed for this thesis, with the ambition of creating a dataset with a more diverse set of ISAs while maintaining the quality of ISAdetect. The dataset contains code sections from compiled open-source libraries. Our main objectives when creating this dataset (which we will discuss in this section) were:

- Ensuring sufficient diversity of ISAs to supplement previous datasets
- Ensuring sufficient quantity of data
- Ensuring that the binary data is representative of production-ready binary code seen in reverse engineering scenarios

BuildCross ultimately contains object code from static libraries for **40** different ISAs. The total dataset size is **123MB**, with a per-architecture average of **3.00 MB** and a median of **2.51 MB**. The dataset sizes and file counts for each architecture are listed in Table A.1. We believe we have successfully created a dataset that offers greater diversity than ISAdetect, incorporating 39 additional ISAs not present there. The dataset is also balanced for both target features, both when looking at the number of samples per class and the number of architectures per class, shown in Table 5.6 and Table 5.7. However, there is some variability in the number of samples per architecture. For instance, `tilegx` has 11,986 samples, while `bpf`, `epiphany`, `ft32`, `msp430`, `rl78`, and `xstormy16` have less than 1,000. For the purpose of answering our research questions and how our CNN models are able to generalize to unseen architectures, we have been more concerned with the total volume of training data, and we are confident that the decision to leave the cross ISA imbalances as is has not significantly impacted results.

Table 5.6: Number of samples per class for endianness in BuildCross.

Endianness	No. 1024 byte samples	Percentage of samples	Architecture count	Percentage of architectures
big	54,186	44.22%	16	40.00%
little	68,356	55.78%	24	60.00%

Table 5.7: Number of samples per class for instruction width type in BuildCross.

Instruction Width Type	No. 1024 byte samples	Percentage of samples	Architecture count	Percentage of architectures
variable	52,367	42.73%	22	55.00%
fixed	70,175	57.27%	18	45.00%

Although our dataset is around six times the size of CpuRec (**20.98 MB** vs. **119.88 MB**) and contains approximately half the number of architectures (**40** compared to **76**), using individual file samples alone would be too limiting for training large deep learning models. We had to resort to file splitting to reach a level of trainable data comparable to ISAdetect, and without the ability to analyze code similarity across the splits, we cannot fully verify the quality of the resulting data. However, our focus on library code may be advantageous, as libraries inherently separate

commonly used functionality into reusable functions, which we believe are likely to create representative samples of binary code. File splitting on BuildCross is a better option than on CpuRec, where some files have duplicated code sections three or four times to fill file size requirements [72]. Nevertheless, we recognize that samples from unique programs would likely provide more distinctive and representative training examples.

We were also concerned that the static library object code might not adequately represent features found in full binary programs. Our decision to compile static libraries instead of executable programs came from two practical challenges: the difficulty of finding compiled programs for rare architectures, and the difficulty in finding source code that could be cross-compiled for our target architectures. While executable compiled programs would have been ideal to represent real-world scenarios, the availability and standalone nature of these open-source libraries made them the only feasible alternative we found for creating a dataset of sufficient size, given the scope of the project.

We investigated the potential impact of using libraries further, and came to the conclusion that object code from static libraries serves as an appropriate representation based on several key findings. The transformation from static library code to a program binary occurs exclusively during the linking stage, as illustrated in Figure 2.3. Static library object code essentially consists of compiled functions and subroutines, which are handled identically to object code from compiled source files in working programs. The machine code itself is identical in nature – the CPU instructions extracted from a static library and placed into the final binary are the same format as instructions from the directly compiled source files. Once linked, they are all sequences of machine instructions. While the linking stage does not affect the machine instructions themselves, the linker does resolve symbol references and memory addresses. Object code typically contains placeholders for these references, which are replaced with the actual addresses during linking. However, the inherent features of the ISA do not change despite unresolved symbols, and the instructions themselves remain valid. While we acknowledge that the linking stage modifies the object code in ways that could potentially influence the CNNs trained on this data, we believe that it does not compromise our research objectives. If CNNs are indeed capable of detecting ISA features (a positive answer to RQ1), then the presence of unresolved symbols and placeholder addresses in library object code should not impair the models' ability to learn the architectural characteristics we aim to identify.

To validate this approach, we attempted to forcefully link the libraries to see if this yielded better results. Our tests revealed that linking all object code into a single binary across different bare metal targets consistently resulted in inflated code sections compared to the original library object code. For example, on the Epiphany architecture shown by the use of its cross-compiler toolchain below this paragraph, the .text section increased substantially in size, from 63 KB to 110 KB. This inflation likely results from the addition of C runtime initialization code (such as `crt0.c`), padding/alignment requirements, and other overhead necessary for bare metal execution. This standardized initialization code would appear nearly identical across all binaries for the same architecture, providing no meaningful differences in features for classification tasks. In fact, the inclusion of such similar boilerplate code would likely hurt classifier performance with code splitting, as the limited input window of our model would capture unique less architecture-specific code, potentially diluting the patterns we aim to detect.

Despite initial concerns about dataset quality, our experiments provided validation of BuildCross's effectiveness. When incorporating BuildCross in the training dataset, we observed

considerable performance improvements when testing on CpuRec – an increase of **8.3 p.p.** in average accuracy for endianness detection and a notable **26.9 p.p.** improvement for instruction width type classification across our best-performing models. These substantial improvements, particularly in instruction width type detection, provide good evidence supporting both the code quality and accuracy of the dataset's labeling. These results suggest that the dataset successfully captures the architectural features necessary for effective classification.

5.6 Sustainability implications and ethical considerations

While this thesis focuses on a specialized technical problem in computer science, our contributions may also have broader implications for sustainability and raise certain ethical concerns. We examine some of these implications and relate them to the United Nations Sustainable Development Goals (SGD) [105].

Our work contributes to the field of reverse engineering. Reverse engineering is a crucial part of malware analysis and digital forensics. SGD 16 targets peace, justice, and strong institutions. Enhanced capabilities in reverse engineering help combat cybersecurity threats, which impacts this goal in a positive way. However, reverse engineering techniques also have the potential for misuse by malicious actors. If reverse engineers with malicious intent discover vulnerabilities in the software, they may use this information to perform illegal activities. While we believe better reverse engineering tools provide a net benefit for software security and thus support SGD 16, there are still considerations to make regarding aiding malicious actors.

Unfortunately, reverse engineering is occasionally used for misusing proprietary software. Malicious actors may steal secrets embedded in compiled code, illegally copy or clone functionality, or bypass licensing mechanisms in the software. As a consequence of this, there is a risk that our work undermines SGD 9, which relates to resilient infrastructure and innovation.

The environmental impact of modern AI tools is commonly criticized. Deep learning models require significant amounts of energy, both during training and during inference. For instance, in 2023, the daily energy usage for inference for ChatGPT was estimated at 564 MWh [106]. This amount of energy could power around 20,000 households, and it is reasonable to assume that the power usage of ChatGPT has increased significantly since this estimation was made. This is a concerning trend that may undermine SGD 12, which calls for responsible consumption and production. In our work, we use CNNs, which is a deep learning architecture with significant computational demands. By sticking to traditional machine learning algorithms and manual feature engineering, significant amounts of energy would likely be spared. However, our results also indicate that very small CNN models perform as well or better than larger and deeper networks. Since the smaller models require far less computational power to train and use, this result will hopefully steer future research toward more energy-efficient approaches.

5.7 Limitations

The most significant limitation of our work with regard to our research questions is that we only consider two target features: endianness and instruction width type. The rationale for this is partly time and resource constraints, but also because other ISA features such as instruction width in bits (for fixed-width ISAs), CISC/RISC type, or number of registers are generally more ambiguous features, and the labeling of these features would likely not be as well-defined as the

features we chose. However, we do acknowledge that this limits our ability to answer RQ1 in a complete manner.

Furthermore, RQ2 is concerned with how the model architecture impacts the results. While we experimented with small, custom CNN architectures and large, pre-defined architectures like ResNet, we did not explore medium-sized CNN models in the range of 1-5 million trainable parameters. Such an exploration might have revealed more insights in performance characteristics and overfitting behavior, potentially identifying model architectures better suited to the task and available data.

Our models are exclusively trained and tested on the code sections of the binary file. The rationale behind this is that the non-code sections, particularly the headers, of a binary file often reveal information about the ISA, and we would risk that the model overfits on this. During the initial exploration phase, we did notice that training on the full binary file resulted in higher model performance across the board, supporting our hypothesis that training on full binary files leads to models learning to identify other ISA-specific patterns in the non-code sections of the file, rather than fitting to inherent ISA features, which is undesirable. The data section of binary files also contains strings and information unrelated to the ISA, potentially introducing more noise. However, since we only train on code sections, the reverse engineer would need to identify the code section of an unknown binary before using our models to detect ISA features. This can be a difficult task for undocumented file formats and instruction sets, and might limit the practical usability of our models.

The lack of interpretability of neural networks is also a limitation of our work. This is a common issue with deep learning models in general. While other machine learning techniques such as linear regression, nearest neighbor, and decision trees are easy to inspect and interpret, neural networks operate in a way that makes the model weights hard to reason about after training. Because of this, we are not able to make strong claims about why the models predict the way they do in our performance analysis. That said, we still provide insights that we consider likely based on observed model behavior.

For the BuildCross dataset specifically, we have identified several limitations that affect our contributions. The two existing datasets (ISAdetect and CpuRec) face their own challenges related to labeling accuracy, architectural diversity, and training data volume, which we have previously discussed in Section 5.5. While our BuildCross dataset addresses some of these limitations, it also introduces new shortcomings. In terms of architectural diversity, we encountered several roadblocks that prevented us from reaching the same level of diversity as CpuRec within the scope of the thesis. The primary challenge was that many older architectures had incomplete GCC toolchains missing C standard library functions, making cross-compiler builds difficult. Additionally, our pipeline relied on ELF header-based tools for feature extraction and code section identification. This meant we could not support targets like `PDP-11` and `68HC08`, which use different file formats. We also discovered that cross-compilers for many supported targets required specific legacy versions of GCC. Pursuing these legacy toolchains was unfavorable from a cost-benefit point of view. Each legacy GCC version required approximately 15GB of storage space and cumbersome manual configuration with long trial-and-error cycles, yet would yield only one to three additional architectures. These older architectures often lacked support for the libraries we needed to cross-compile, creating a high risk that the setup effort would not be worth it. Ultimately, we prioritized focusing on architectures supported by the base configuration of the script that could reliably generate the training data needed for our research objectives.

Another limitation of the BuildCross cross-compiler suite is its reliance on the work of Mikael Pettersson [77]. Even though his buildcross.sh script was tweaked by us to fit our needs, a lot of the trial and error work of getting different GCC, binutils, and libc versions and implementations to work together was done by him, and some architectures required specific patches that are hosted on his GitHub. This means that the BuildCross dataset is dependent on external sources, which may not be available in the future. We have made our code available on GitHub, but we cannot guarantee that the external toolchain will remain available indefinitely. This could limit the reproducibility of our results in the future and is something we regret not having the time to address.

Chapter 6

Conclusion

In this thesis, we have investigated the application of Convolutional Neural Networks (CNNs) for detecting Instruction Set Architecture (ISA) features from raw binary code, to address the challenge of reverse engineering binaries from unknown or undocumented architectures. Our work has explored a new direction compared to prior literature, leveraging deep learning models' ability to automatically extract meaningful patterns from the input rather than relying on manual feature engineering.

RQ1 aimed to identify which ISA features were suitable for detection through CNN-based approaches. Our experiments showed that for our two tested target features – endianness and instruction width type – the models performed similarly for both features. Using Leave-One-Group-Out Cross-Validation (LOGO CV) on the ISAdetect dataset, we observed accuracies up to 90.3% and 88.0% for endianness and instruction width type classification, respectively. When we extended the evaluation with more datasets, we saw that accuracy for both target features dropped to less than 75% on previously unseen architectures. While we observed a notable decline in performance when evaluating on a large set of unseen ISAs, we do note that the model performance was on par with prior research that relied on carefully engineered features.

RQ2 asked whether the way we encoded the binary files for the CNN input layer impacts the CNNs' ability to learn ISA characteristics. Our results revealed that while both one-dimensional and two-dimensional encodings proved viable, neither was consistently better in all scenarios. For endianness detection, the dimensionality of the encoding appeared to have minimal impact, likely because endianness patterns manifest at the byte level regardless of spatial arrangement. Instruction width type detection showed slightly higher accuracies with two-dimensional encodings in some experiments, possibly due to the repeating patterns in some fixed-width instruction sets. However, the performance difference was not large enough to claim a statistically significant advantage for two-dimensional encodings.

RQ3, which explored different CNN architectures and compared their performance, led us to conclude that large models with many parameters do not exhibit better performance than smaller models for this task. We saw that in nearly all experiments, the small models with less than 150,000 trainable parameters performed on par or better than the large ResNet-50 model with 23.5 million parameters. Additionally, we observed that embedding layers seemed to have a positive effect. In many experiments, particularly for endianness detection, the embedding-augmented

models showed statistically significant performance improvements over their non-embedding counterparts.

In conclusion, our work has shown that CNN-based approaches to detecting individual ISA features perform on par with, but not significantly better than, existing approaches in prior research. However, there is value in the automatic feature engineering characteristics of deep learning models over the traditional machine learning approaches due to the elimination of extensive feature engineering efforts. Through this thesis, we have demonstrated that very small CNNs work just as well as larger ones for this particular task, and have also demonstrated the effectiveness of embedding layers when applying deep learning techniques to binary code analysis. Lastly, our research has highlighted the importance of high-quality, well-labeled datasets for deep learning applications in binary reverse engineering, and the development of the BuildCross dataset has not only enhanced our own research capabilities but will also provide a valuable resource for future work in this field.

6.1 Future work

We identify several possibilities for building on our work. Firstly, extending our approach to more target features would allow for a deeper understanding of the binary file. Considering that CNN-based models do not demand extensive feature engineering, applying the same methodology for other ISA features, such as word size, instruction width (for fixed-width instruction sets), or register count, would be feasible given a dataset with clearly defined labels.

Furthermore, extending our approach to operate on full binary files, rather than just code sections, would be beneficial. This would enable our method to be applied even when the code section of a binary cannot be easily identified. To achieve this, we propose adopting a "rolling window" technique, as demonstrated in a previous binary analysis study by Beckman & Haile [47].

We also reason that there are more applications for which CNNs can be applied to binary analysis. The application of CNNs to closed-set ISA detection is a convincing direction for future research, building upon prior work in this area (see Section 2.4.1). In our experiments, we observed that CNN models tended to overfit on ISA-specific characteristics. While causing poor generalizability when detecting individual features from an unseen ISA, this tendency could be favorable for classifying a binary's ISA from a known set. Our initial investigation suggests that the model architectures used in this thesis can be effectively repurposed for full ISA detection, demonstrating high accuracy in this task. Using a CNN model trained on both ISAdetect and the diverse BuildCross dataset developed in this thesis, there is potential for advancing the state of the art in ISA detection.

Another potential application of CNNs is automatic identification of code sections within raw binary files. Usually, the reverse engineering process requires knowing which parts of the binary file actually contain CPU instructions. Techniques from image segmentation, such as the U-Net architecture, could be adapted for this purpose [107]. A one-dimensional CNN capable of segmentation could be trained to predict a segmentation mask, highlighting the regions of the binary file corresponding to executable code.

Finally, further work could be carried out to extend our custom dataset. BuildCross was created to address the gaps in ISAdetect and CpuRec, with architectures deliberately chosen to avoid over-

lap with ISAdetect. While this design decision was based on the requirements of this thesis, BuildCross could be expanded for future research. The dataset could incorporate well-established architectures such as ARM, MIPS, PowerPC, and x86 to provide broader coverage of mainstream instruction sets. Additionally, numerous uncommon architectures supported by previous versions of the GNU Compiler Collection (GCC) toolchain were excluded due to time constraints. These could be added to create an even more comprehensive and robust dataset for future binary analysis research.

References

[1] Z. Chen, B. Pan, and Y. Sun, "A survey of software reverse engineering applications," in *Artificial intelligence and security*, X. Sun, Z. Pan, and E. Bertino, Eds., Cham: Springer International Publishing, 2019, pp. 235–245.

[2] J. Ledin and D. Farley, *Modern computer architecture and organization: Learn x86, ARM, and RISC-v architectures and the design of smartphones, PCs, and cloud servers*. Packt Publishing Ltd, 2022.

[3] C. Spensky, "Analyzing and securing embedded systems," PhD thesis, UC Santa Barbara, 2020.

[4] J. Kinder, "Towards static analysis of virtualization-obfuscated binaries," in *2012 19th working conference on reverse engineering*, Oct. 2012, pp. 61–70. doi: 10.1109/WCRE.2012.16.

[5] M. Liang, Z. Li, Q. Zeng, and Z. Fang, "Deobfuscation of virtualization-obfuscated code through symbolic execution and compilation optimization," in *Information and communications security: 19th international conference, ICICS 2017, beijing, china, december 6-8, 2017, proceedings 19*, Springer, 2018, pp. 313–324.

[6] A. Costin and J. Zaddach, "Iot malware: Comprehensive survey, analysis framework and case studies," *BlackHat USA*, vol. 1, no. 1, pp. 1–9, 2018.

[7] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era—a state of the art survey," *ACM Comput. Surv.*, vol. 52, no. 5, Sep. 2019, doi: 10.1145/3329786.

[8] S. J. Sulebak and M. Svartveit, "Machine learning for reverse engineering & convolutional neural networks for binary code analysis: A systematic literature review," 2024. Available: https://mikkelsvartveit.github.io/thesis/preproject.pdf

[9] A. Chernov and K. Troshina, "Reverse engineering of binary programs for custom virtual machines," in *ReCon 2012*, 2012. Available: https://recon.cx/2012/schedule/attachments/40_Chernov-Troshina.pdf

[10] Advanced RISC Machines Ltd, *ARM7 data sheet*. Advanced RISC Machines Ltd, 1994. Accessed: Jun. 05, 2025. [Online]. Available: https://developer.arm.com/documentation/ddi0027/latest/

[11] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, "Architecture of the IBM system/360," *IBM Journal of Research and Development*, vol. 8, no. 2, pp. 87–101, 1964.

[12] IBM Corporation, "The IBM system/360." IBM Corporation, 2024. Accessed: Jun. 05, 2025. [Online]. Available: https://www.ibm.com/history/system-360

[13] Wikipedia contributors, "Word (computer architecture)." Wikimedia Foundation, 2025. Accessed: Jun. 05, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Word_(computer_architecture)

[14] MIPS Technologies, Inc., *MIPS32™ architecture for programmers volume II: The MIPS32™ instruction set*. MIPS Technologies, Inc., 2001. Accessed: Dec. 28, 2024. [Online]. Available: https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol2.pdf

[15] Motorola Inc. and IBM Microelectronics, *PowerPC 604 RISC microprocessor technical summary*. Motorola Inc., 1994. Accessed: Dec. 28, 2024. [Online]. Available: https://www.nxp.com/docs/en/data-sheet/MPC604.pdf

[16] IBM Corporation, "The IBM system/360." IBM Corporation, 2024. Accessed: Jun. 05, 2025. [Online]. Available: https://www.ibm.com/history/fortran

[17] D. M. Ritchie, "The development of the c language," *ACM Sigplan Notices*, vol. 28, no. 3, pp. 201–208, 1993.

[18] OSDev contributors, "Why do i need a cross compiler?" OSDev Wiki. Accessed: Jun. 05, 2025. [Online]. Available: https://wiki.osdev.org/Why_do_I_need_a_Cross_Compiler%3F

[19] Kitware, Inc., "Cross compiling with CMake." Kitware, Inc. CMake Documentation. Accessed: Jun. 05, 2025. [Online]. Available: https://cmake.org/cmake/help/book/mastering-cmake/chapter/Cross%20Compiling%20With%20CMake.html

[20] Free Software Foundation contributors, "GCC wiki." Free Software Foundation; GCC Wiki. Accessed: Jun. 05, 2025. [Online]. Available: https://gcc.gnu.org/wiki

[21] Free Software Foundation contributors, "A brief history of GCC." Free Software Foundation; GCC Wiki. Accessed: Jun. 05, 2025. [Online]. Available: https://gcc.gnu.org/wiki/History

[22] Free Software Foundation contributors, "GNU binutils." Free Software Foundation; GNU Binutils. Accessed: Jun. 05, 2025. [Online]. Available: https://www.gnu.org/software/binutils/

[23] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990, doi: 10.1109/52.43044.

[24] E. Fauzi, B. Hendradjaya, and W. D. Sunindyo, "Reverse engineering of source code to sequence diagram using abstract syntax tree," *Proceedings of 2016 International Conference on Data and Software Engineering, ICoDSE 2016*. 2017. doi: 10.1109/ICODSE.2016.7936137.

[25] H. Müller and H. Kienle, "A small primer on software reverse engineering," *Technical Report, University of Victoria*, Jan. 2009.

[26] A. Qasem, P. Shirani, M. Debbabi, L. Wang, B. Lebel, and B. L. Agba, "Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies," *ACM Computing Surveys*, vol. 54, no. 2, 2022, doi: 10.1145/3432893.

[27] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2019–May. pp. 472–489, 2019. doi: 10.1109/SP.2019.00003.

[28] K. P. Subedi, D. R. Budhathoki, and D. Dasgupta, "Forensic analysis of ransomware families using static and dynamic analysis," *Proceedings - 2018 IEEE Symposium on Security and Privacy Workshops, SPW 2018*. pp. 180–185, 2018. doi: 10.1109/SPW.2018.00033.

[29] D. Votipka, S. M. Rabin, K. Micinski, J. S. Foster, and M. M. Mazurek, "An observational investigation of reverse engineers' processes," *Proceedings of the 29th USENIX Security Symposium*. pp. 1875–1892, 2020. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85091828567&partnerID=40&md5=16d9d8131667a5095d5b391e538b7181

[30] Y. Shoshitaishvili *et al.*, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE symposium on security and privacy*, 2016.

[31] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16–21–November–2014. pp. 389–400, 2014. doi: 10.1145/2635868.2635900.

[32] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," *16th USENIX Security Symposium*. pp. 275–290, 2007. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85045981068&partnerID=40&md5=5ce01c3949e171308aca4aaab5593f72

[33] Hex-Rays, "IDA pro." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://hex-rays.com/ida-pro/

[34] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Angr." Available at https://github.com/angr/angr, 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://angr.io/

[35] N. A. Görke and M. H. Steensland, "A semi-systematic review of reverse engineering: Processes, tools, and their internal operations," Unpublished work, Dec. 2024.

[36] S. Kairajärvi, A. Costin, and T. Hämäläinen, "ISAdetect: Usable automated detection of CPU architecture and endianness for executable binary files and object code," *CODASPY 2020 - Proceedings of the 10th ACM Conference on Data and Application Security and Privacy*. pp. 376–380, 2020. doi: 10.1145/3374664.3375742.

[37] P. De Nicolao, M. Pogliani, M. Polino, M. Carminati, D. Quarta, and S. Zanero, "ELISA: ELiciting ISA of raw binaries for fine-grained code and data separation," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10885 LNCS, pp. 351–371, 2018, doi: 10.1007/978-3-319-93411-2_16.

[38] National Security Agency, "Ghidra." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://ghidra-sre.org/

[39] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *Proceedings of the 2nd ACM SIGPLAN program protection and reverse engineering workshop*, in PPREW '13. New York, NY, USA: Association for Computing Machinery, 2013. doi: 10.1145/2430553.2430558.

[40] C. Collberg, "The tigress c obfuscator." University of Arizona, 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://tigress.wtf/index.html

[41] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st international workshop on software protection, SPRO'15, firenze, italy, may 19th, 2015*, B. Wyseur, Ed., IEEE, 2015, pp. 3–9. doi: 10.1109/SPRO.2015.10.

[42] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998, doi: 10.1109/5.726791.

[43] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition." 2015. Available: https://arxiv.org/abs/1409.1556

[44] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space." 2013. Available: https://arxiv.org/abs/1301.3781

[45] J. Andreassen and D. Morrison, "Discovery of endianness and instruction size characteristics in binary programs from unknown instruction set architectures," *Norsk IKT-konferanse for forskning og utdanning*, no. 1, 2024, Available: https://www.ntnu.no/ojs/index.php/nikt/article/view/6225

[46] J. Clemens, "Automatic classification of object code using machine learning," *Proceedings of the Digital Forensic Research Conference, DFRWS 2015 USA*. pp. S156–S162, 2015. doi: 10.1016/j.diin.2015.05.007.

[47] B. Beckman and J. Haile, "Binary analysis with architecture and code section detection using supervised machine learning," *Proceedings - 2020 IEEE Symposium on Security and Privacy Workshops, SPW 2020*. pp. 152–156, 2020. doi: 10.1109/SPW50608.2020.00041.

[48] Y. Ma, L. Han, H. Ying, S. Yang, W. Zhao, and Z. Shi, "SVM-based instruction set identification for grid device firmware," *Proceedings of 2019 IEEE 8th Joint International Information Technology and Artificial Intelligence Conference, ITAIC 2019*. pp. 214–218, 2019. doi: 10.1109/ITAIC.2019.8785564.

[49] D. Sahabandu, J. S. Mertoguno, and R. Poovendran, "A natural language processing approach for instruction set architecture identification," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 4086–4099, 2023, doi: 10.1109/TIFS.2023.3288456.

[50] A. Panconesi, Marian, W. Cukierski, and W. B. -Cup Committee, "Microsoft malware classification challenge (BIG 2015)." https://kaggle.com/competitions/malware-classification, 2015.

[51] R. K. Rahul, T. Anjali, V. K. Menon, and K. P. Soman, "Deep learning for network flow analysis and malware classification," in *Security in computing and communications*, S. M. Thampi, G. Martínez Pérez, C. B. Westphall, J. Hu, C. I. Fan, and F. Gómez Mármol, Eds., Singapore: Springer Singapore, 2017, pp. 226–235.

[52] M. Kumari, G. Hsieh, and C. A. Okonkwo, "Deep learning approach to malware multi-class classification using image processing techniques," in *2017 international conference on computational science and computational intelligence (CSCI)*, Dec. 2017, pp. 13–18. doi: 10.1109/CSCI.2017.3.

[53] C. Yang *et al.*, "A convolutional neural network based classifier for uncompressed malware samples," in *Proceedings of the 1st workshop on security-oriented designs of computer architectures and processors*, in SecArch'18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 15–17. doi: 10.1145/3267494.3267496.

[54] M. Khan, D. Baig, U. S. Khan, and A. Karim, "Malware classification framework using convolutional neural network," in *2020 international conference on cyber warfare and security (ICCWS)*, Oct. 2020, pp. 1–7. doi: 10.1109/ICCWS48432.2020.9292384.

[55] S. Sartoli, Y. Wei, and S. Hampton, "Malware classification using recurrence plots and deep neural network," in *2020 19th IEEE international conference on machine learning and applications (ICMLA)*, Dec. 2020, pp. 901–906. doi: 10.1109/ICMLA51294.2020.00147.

[56] B. Prima and M. Bouhorma, "TRANSFER LEARNING AND SMOTE ALGORITHM FOR IMAGE-BASED MALWARE CLASSIFICATION," in *Proceedings of the 4th international conference on networking, information systems & security*, in NISS '21. New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3454127.3457631.

[57] J. Liang, Z. Ning, Y. Zhou, and D. Cao, "Fine-grained classification of malicious code based on CNN and multi-resolution feature fusion," in *2021 6th international conference on computational intelligence and applications (ICCIA)*, 2021, pp. 123–127. doi: 10.1109/ICCIA52886.2021.00031.

[58] I. Alam, M. Samiullah, U. Kabir, S. Woo, C. K. Leung, and H. H. Nguyen, "SREMIC: Spatial relation extraction-based malware image classification," in *2024 18th international conference on ubiquitous information management and communication (IMCOM)*, Jan. 2024, pp. 1–8. doi: 10.1109/IMCOM60618.2024.10418339.

[59] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: Visualization and automatic classification," in *Proceedings of the 8th international symposium on visualization for cyber security*, in VizSec '11. New York, NY, USA: Association for Computing Machinery, 2011. doi: 10.1145/2016904.2016908.

[60] I. García-Daza-Cervantes, R. Reyes-Reyes, C. Cruz-Ramos, V. Ponomaryov, and D. Ponomaryov, "Malware classification using distance and directional local binary patterns," in *2019 IEEE international scientific-practical conference problems of infocommunications, science and technology (PIC s&t)*, Oct. 2019, pp. 397–401. doi: 10.1109/PICST47496.2019.9061336.

[61] W. El-Shafai, I. Almomani, and A. AlKhayer, "Visualized malware multi-classification framework using fine-tuned CNN-based transfer learning models," *Applied Sciences*, vol. 11, no. 14, 2021, doi: 10.3390/app11146446.

[62] X. Li, X. Li, F. Wang, W. Li, and A. Li, "A malware detection method based on machine learning and ensemble of regression trees," in *2021 2nd international conference on artificial intelligence and information systems*, in ICAIIS 2021. New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3469213.3470713.

[63] T. T. Son, C. Lee, H. Le-Minh, N. Aslam, and V. C. Dat, "An enhancement for image-based malware classification using machine learning with low dimension normalized input images," *Journal of Information Security and Applications*, vol. 69, p. 103308, 2022, doi: https://doi.org/10.1016/j.jisa.2022.103308.

[64] B. T. Hammad, N. Jamil, I. T. Ahmed, Z. M. Zain, and S. Basheer, "Robust malware family classification using effective features and classifiers," *Applied Sciences*, vol. 12, no. 15, 2022, doi: 10.3390/app12157877.

[65] A. S. Parihar, S. Kumar, and S. Khosla, "S-DCNN: Stacked deep convolutional neural networks for malware classification," *Multimedia Tools and Applications*, vol. 81, no. 21, pp. 30997–31015, 2022, doi: 10.1007/s11042-022-12615-7.

[66] B. Al-Masri, N. Bakir, A. El-Zaart, and K. Samrouth, "Dual convolutional malware network (DCMN): An image-based malware classification using dual convolutional neural networks," *Electronics*, vol. 13, no. 18, 2024, doi: 10.3390/electronics13183607.

[67] S. R. B. Alvee, B. Ahn, T. Kim, Y. Su, Y. Youn, and M. Ryu, "Ransomware attack modeling and artificial intelligence-based ransomware detection for digital substations," in *2021 6th IEEE workshop on the electronic grid (eGRID)*, Nov. 2021, pp. 01–05. doi: 10.1109/eGRID52793.2021.9662158.

[68] S. Yang, Z. Shi, G. Zhang, M. Li, Y. Ma, and L. Sun, "Understand code style: Efficient CNN-based compiler optimization recognition system," in *ICC 2019 - 2019 IEEE international conference on communications (ICC)*, May 2019, pp. 1–6. doi: 10.1109/ICC.2019.8761073.

[69] D. Pizzolotto and K. Inoue, "Identifying compiler and optimization level in binary code from multiple architectures," *IEEE Access*, vol. 9, pp. 163461–163475, 2021, doi: 10.1109/ACCESS.2021.3132950.

[70] S. Kairajärvi and A. Costin, "ISAdetect binary file and object code dataset." http://urn.fi/urn:nbn:fi:att:d58324bd-1cf9-49cf-99cd-5bc2ba781e38, Mar. 2020.

[71] L. Granboulan, "Cpu_rec_sstic_english.md," *GitHub repository*. GitHub, Feb. 2020. Accessed: Feb. 13, 2025. [Online]. Available: https://github.com/airbus-seclab/cpu_rec/blob/master/doc/cpu_rec_sstic_english.md

[72] L. Granboulan, "Cpu_rec_corpus," *GitHub repository*. GitHub. Accessed: Feb. 13, 2025. [Online]. Available: https://github.com/airbus-seclab/cpu_rec/tree/master/cpu_rec_corpus

[73] M. Svartveit and S. Sulebak, "Thesis," *GitHub repository*. GitHub. Accessed: Jun. 06, 2025. [Online]. Available: https://github.com/mikkelsvartveit/thesis

[74] NTNU, "Idun – high performance computing." https://www.hpc.ntnu.no/idun/.

[75] P. Golik, P. Doetsch, and H. Ney, "Cross-entropy vs. Squared error training: A theoretical and experimental comparison," in *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, Aug. 2013, pp. 1756–1760. doi: 10.21437/Interspeech.2013-436.

[76] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization." 2019. Available: https://arxiv.org/abs/1711.05101

[77] M. Pettersson, "Buildcross," *GitHub repository*. GitHub. Accessed: Feb. 22, 2025. [Online]. Available: https://github.com/mikpe/buildcross

[78] Singularity Developers, *Singularity*. (Apr. 2021). Zenodo. doi: 0.5281/zenodo.1310023.

[79] Singularity Developers, "Apptainer," *GitHub repository*. GitHub. Accessed: Jun. 04, 2025. [Online]. Available: https://github.com/apptainer/apptainer

[80] Kitware, Inc., "CMake - Cross Platform Make." 2025. Accessed: Jun. 04, 2025. [Online]. Available: https://cmake.org/

[81] FreeType Project, "FreeType source code." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://gitlab.freedesktop.org/freetype/freetype

[82] FreeType Project, "FreeType." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://www.freetype.org/

[83] libgit2 contributors, "libgit2 source code." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://github.com/libgit2/libgit2

[84] libgit2 contributors, "libgit2." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://libgit2.org/

[85] libjpeg-turbo Project, "Libjpeg-turbo source code." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://github.com/libjpeg-turbo/libjpeg-turbo

[86] libjpeg-turbo Project, "Libjpeg-turbo." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://libjpeg-turbo.org/

[87] The PNG Development Group, "Libpng source code." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://github.com/pnggroup/libpng

[88] The PNG Development Group, "Libpng." 2025. Accessed: Feb. 13, 2025. [Online]. Available: http://www.libpng.org/pub/png/libpng.html

[89] Google Inc., "Libwebp source code." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://chromium.googlesource.com/webm/libwebp/

[90] Google Inc., "Libwebp." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://developers.google.com/speed/webp

[91] K. Simonov and I. döt Net, "LibYAML source code." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://github.com/yaml/libyaml

[92] K. Simonov and I. döt Net, "LibYAML." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://pyyaml.org/wiki/LibYAML

[93] P. Hazel and U. of Cambridge, "PCRE2 source code." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://github.com/PCRE2Project/pcre2

[94] P. Hazel and U. of Cambridge, "PCRE2." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://www.pcre.org/

[95] L. Collin, xz-utils authors, and contributors, "XZ utils source code." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://git.tukaani.org/?p=xz.git

[96] L. Collin and xz-utils contributors, "XZ utils." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://tukaani.org/xz/

[97] J. Gailly and M. Adler, "Zlib source code." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://github.com/madler/zlib

[98] G. Roelof, J. Gailly, and M. Adler, "Zlib." 2025. Accessed: Feb. 13, 2025. [Online]. Available: https://www.zlib.net/

[99] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition." 2015. Available: https://arxiv.org/abs/1512.03385

[100] scikit-learn, "MLPClassifier — scikit-learn 1.6.1 documentation." 2025. Accessed: May 15, 2025. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

[101] M. Abdelhamid and A. Desai, "Balancing the scales: A comprehensive study on tackling class imbalance in binary classification." 2024. Available: https://arxiv.org/abs/2409.19751

[102] Google for Developers, "Datasets: Imbalanced datasets." Google; https://developers.google.com/machine-learning/crash-course/overfitting/imbalanced-datasets, 2025.

[103] M. Girkar, P. Anvin, H. Lu, D. Shkurko, and V. Zakharin, "The x32 ABI: A new software convention for performance on intel 64 processors," *Intel Technology Journal*, vol. 16, pp. 114–128, 2012, Available: https://www.intel.com/content/dam/www/public/us/en/documents/research/2012-vol16-iss-1-intel-technology-journal.pdf

[104] K. E. Plambeck, W. Eckert, R. R. Rogers, and C. F. Webb, "Development and attributes of z/architecture," *IBM Journal of Research and Development*, vol. 46, no. 4.5, pp. 367–379, 2002, doi: 10.1147/rd.464.0367.

[105] United Nations General Assembly, "Transforming our world: The 2030 agenda for sustainable development." 2015. Available: https://sdgs.un.org/2030agenda

[106] A. de Vries, "The growing energy footprint of artificial intelligence," *Joule*, vol. 7, no. 10, pp. 2191–2194, 2023, doi: https://doi.org/10.1016/j.joule.2023.09.004.

[107] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation." 2015. Available: https://arxiv.org/abs/1505.04597

Appendix A

Dataset additional information

Contains additional information about the datasets used in this thesis. This includes information about the dataset sizes, number of samples, labeling, as well as comparisons with related work.

A.1 BuildCross dataset label list and sizes

The BuildCross dataset contains a total of **119.88 MB** of data, with an average size of **3.00 MB** per ISA and a median size of **2.51 MB** per ISA. The dataset is divided into 40 different ISAs, each with a varying number of files and samples. All files smaller than 1024 bytes are ignored.

Table A.1: The list of labels used in the BuildCross dataset. The labels are based on the ELF headers of the generated code and the disassembly of the binaries

ISA	Endianness	Instruction Width Type	Total Size (MB)	Number of Files	No. 1024 byte sized samples
arc	little	variable	3.23	14	3299
arceb	big	variable	1.70	12	1731
bfin	little	variable	2.88	14	2942
bpf	little	fixed	0.02	1	19
c6x	big	fixed	5.55	8	5679
cr16	little	variable	1.97	13	2012
cris	little	variable	3.98	14	4074
csky	little	variable	4.15	14	4247
epiphany	little	variable	0.46	6	471
fr30	big	variable	2.17	7	2223
frv	big	fixed	4.93	14	5037
ft32	little	fixed	0.44	9	440
h8300	big	variable	4.30	9	4402
iq2000	big	fixed	2.41	8	2466
kvx	little	variable	4.90	14	5016
lm32	big	fixed	3.32	13	3396
loongarch64	little	fixed	4.71	14	4818
m32r	big	fixed	1.96	12	1997
m68k-elf	big	variable	1.83	12	1866

ISA	Endianness	Instruction Width Type	Total Size (MB)	Number of Files	No. 1024 byte sized samples
mcore	little	fixed	1.24	7	1270
mcoreeb	big	fixed	1.24	7	1270
microblaze	big	fixed	5.74	14	5867
microblazeel	little	fixed	5.71	14	5840
mmix	big	fixed	4.22	13	4314
mn10300	little	variable	1.70	12	1732
moxie	big	variable	2.19	12	2237
moxieel	little	variable	2.19	12	2232
msp430	little	variable	0.42	5	432
nds32	little	variable	2.85	14	2908
nios2	little	fixed	4.21	14	4301
or1k	big	fixed	5.42	14	5544
pru	little	fixed	2.39	8	2443
rl78	little	variable	0.63	5	643
rx	little	variable	1.46	12	1486
tilegx	little	fixed	11.71	14	11986
tricore	little	variable	1.61	8	1646
v850	little	variable	3.53	10	3609
visium	big	fixed	3.41	12	3488
xstormy16	little	variable	0.48	5	490
xtensa	big	variable	2.61	14	2669

A.2 ISAdetect dataset label list and sizes

Table A.2: The list of labels and architecture data sizes in the ISAdetect dataset. The labels are based on the labeling by Kairajärvi et al. [70]

Architecture	Endianness	Instruction Width Type	Total Size (MB)	Number of Files
alpha	little	fixed	925.77	3952
amd64	little	variable	564.43	4059
arm64	little	fixed	418.50	3518
armel	little	fixed	466.91	3814
armhf	little	fixed	331.34	3674
hppa	big	fixed	940.76	4830
i386	little	variable	519.50	4484
ia64	little	variable	2044.75	4983
m68k	big	variable	684.06	4313
mips	big	fixed	545.13	3547
mips64el	little	fixed	1117.75	4280
mipsel	little	fixed	545.68	3693
powerpc	big	fixed	547.82	3618
powerpcspe	big	fixed	790.11	3922
ppc64	big	fixed	771.06	2822

Architecture	Endianness	Instruction Width Type	Total Size (MB)	Number of Files
ppc64el	little	fixed	574.67	3521
riscv64	little	fixed	605.14	4285
s390	big	variable	360.46	5118
s390x	big	variable	532.86	3511
sh4	little	fixed	723.25	5854
sparc	big	fixed	362.99	4923
sparc64	big	fixed	844.07	3205
x32	little	variable	719.76	4059

A.3 CpuRec dataset label list and sizes

Table A.3: The list of labels used in the CpuRec dataset. The labels are based on previous work by [45], searching online for ISA documentation and disassembly from the BuildCross suite.

Architecture	Endianness	Instruction Width Type	Total Size (KB)
6502	little	variable	6.57
68HC08	big	variable	18.39
68HC11	big	variable	25.17
8051	unknown	variable	15.76
ARC32eb	big	variable	46.09
ARC32el	little	variable	45.88
ARM64	little	fixed	345.32
ARMeb	big	fixed	896.44
ARMel	little	fixed	329.23
ARMhf	little	fixed	230.78
ARcompact	little	variable	118.12
AVR	unknown	variable	193.40
Alpha	little	fixed	1065.17
AxisCris	little	variable	61.11
Blackfin	little	variable	104.82
CLIPPER	little	variable	1059.47
Cell-SPU	unknown	unknown	290.22
CompactRISC	little	variable	56.58
Cray	unknown	variable	1120.00
Epiphany	little	variable	69.03
FR-V	big	fixed	175.25
FR30	big	fixed	141.42
FT32	little	fixed	179.18
H8-300	big	variable	163.47
H8S	unknown	variable	81.52
HP-Focus	unknown	variable	408.00
HP-PA	big	fixed	1057.03
IA-64	little	variable	423.41

Architecture	Endianness	Instruction Width Type	Total Size (KB)
IQ2000	big	fixed	178.65
M32C	little	variable	173.75
M32R	big	fixed	121.89
M68k	big	variable	728.19
M88k	big	fixed	351.18
MCore	little	fixed	101.30
MIPS16	unknown	fixed	95.62
MIPSeb	big	fixed	747.85
MIPSel	little	fixed	425.16
MMIX	big	fixed	387.71
MN10300	little	variable	114.29
MSP430	little	variable	301.51
Mico32	big	fixed	163.39
MicroBlaze	big	fixed	192.71
Moxie	big	variable	140.64
NDS32	little	variable	94.04
NIOS-II	little	fixed	139.11
PDP-11	unknown	variable	124.00
PIC10	unknown	fixed	8.89
PIC16	unknown	fixed	39.16
PIC18	unknown	fixed	45.89
PIC24	little	fixed	82.67
PPCeb	big	fixed	403.82
PPCel	little	fixed	462.20
RISC-V	little	fixed	69.24
RL78	little	variable	337.46
ROMP	big	variable	440.00
RX	little	variable	87.12
S-390	big	variable	453.77
SPARC	big	fixed	1376.51
STM8	unknown	variable	15.35
Stormy16	little	variable	138.34
SuperH	little	fixed	876.43
TILEPro	unknown	variable	112.16
TLCS-90	unknown	variable	23.18
TMS320C2x	unknown	variable	44.94
TMS320C6x	unknown	fixed	105.53
TriMedia	unknown	unknown	462.70
V850	little	variable	132.65
VAX	little	variable	318.00
Visium	big	fixed	274.00
WE32000	unknown	unknown	326.32
X86	little	variable	396.49
X86-64	little	variable	375.41
Xtensa	unknown	variable	87.56

Architecture	Endianness	Instruction Width Type	Total Size (KB)
XtensaEB	big	variable	66.03
Z80	little	variable	20.86
i860	unknown	fixed	598.00

A.4 Dataset labeling comparison with Andreassen

Table A.4: ISAdetect labeling differences between our research and what was presented in Andreassen's paper. Differences highlighted in bold.

ISA	Our endianness	Andreassen endianness	Our instruction width type	Andreassen instruction width type
alpha	little	little	fixed	fixed
amd64	little	little	variable	variable
arm64	little	little	fixed	fixed
armel	little	little	fixed	fixed
armhf	little	little	fixed	fixed
hppa	big	big	fixed	fixed
i386	little	little	variable	variable
ia64	little	little	**variable**	**fixed**
m68k	big	big	**variable**	**unk**
mips	big	big	fixed	fixed
mips64el	little	little	fixed	fixed
mipsel	little	little	fixed	fixed
powerpc	big	big	fixed	fixed
powerpcspe	big	big	fixed	fixed
ppc64	big	big	**fixed**	**unk**
ppc64el	little	little	**fixed**	**unk**
riscv64	little	little	fixed	fixed
s390	big	big	**variable**	**unk**
s390x	big	big	**variable**	**unk**
sh4	**little**	**BI**	**fixed**	**unk**
sparc	big	big	fixed	fixed
sparc64	big	big	fixed	fixed
x32	little	little	**variable**	**unk**

Table A.5: CpuRec labeling differences between our research and what was presented in Andreassen's paper. Differences highlighted in bold. 78k was not in the corpus at the time of downloading the dataset.

ISA	Our Endianness	Andreassen Endianness	Our instruction width type	Andreassen instruction width type
6502	little	little	variable	variable
68HC08	big	big	variable	variable
68HC11	big	big	variable	variable

ISA	Our Endianness	Andreassen Endianness	Our instruction width type	Andreassen instruction width type
78k	-		-	
8051	**na**	**little**	variable	variable
Alpha	little	little	fixed	fixed
ARCompact	little	little	variable	variable
ARM64	little	little	fixed	fixed
ARMeb	big	big	fixed	fixed
ARMel	little	little	fixed	fixed
ARMhf	little	little	fixed	fixed
AVR	**na**	**little**	variable	variable
AxisCris	little	little	**variable**	**fixed**
Blackfin	little	little	variable	variable
Cell-SPU	**bi**	**big**	**unk**	**fixed**
CLIPPER	little	little	variable	variable
CompactRISC	little	little	**variable**	**fixed**
Cray	na	NA	variable	
Epiphany	little	little	variable	variable
FR-V	big	NA	fixed	
FR30	big	big	fixed	fixed
FT32	little	NA	fixed	
H8-300	big	big	variable	variable
H8S	unk	big	variable	
HP-Focus	na	NA	variable	
HP-PA	big	big	fixed	fixed
i860	bi	BI	fixed	
IA-64	little	little	**variable**	**fixed**
IQ2000	big	big	fixed	
M32C	little	NA	variable	
M32R	**big**	**BI**	**fixed**	**variable**
M68k	big	big	variable	
M88k	**big**	**BI**	fixed	fixed
MCore	**little**	**big**	fixed	fixed
Mico32	big	big	fixed	fixed
MicroBlaze	**big**	**BI**	fixed	fixed
MIPS16	bi	BI	fixed	fixed
MIPSeb	big	big	fixed	fixed
MIPSel	little	little	fixed	fixed
MMIX	big	big	fixed	fixed
MN10300	little	little	variable	
Moxie	**big**	**BI**	variable	variable
MSP430	little	little	variable	
NDS32	**little**	**BI**	variable	variable
NIOS-II	little	little	fixed	fixed
PDP-11	**middle**	**little**	**variable**	**fixed**

ISA	Our Endianness	Andreassen Endianness	Our instruction width type	Andreassen instruction width type
PIC10	**na**	**little**	fixed	
PIC16	**na**	**little**	fixed	
PIC18	**na**	**little**	fixed	
PIC24	little	little	fixed	fixed
PPCeb	big	big	fixed	
PPCel	little	little	fixed	
RISC-V	little	little	fixed	fixed
RL78	little	little	variable	
ROMP	big	big	variable	variable
RX	little	little	variable	
S-390	big	big	variable	
SPARC	big	big	fixed	fixed
STM8	na		variable	
Stormy16	little	little	variable	
SuperH	**little**	**BI**	fixed	
TILEPro	unk		variable	
TLCS-90	unk		variable	
TMS320C2x	unk		variable	
TMS320C6x	**unk**	**BI**	fixed	
TriMedia	unk		unk	
V850	little		variable	
Visium	big		fixed	
WE32000	unk		unk	
X86-64	little	little	variable	variable
X86	little	little	variable	variable
Xtensa	bi	BI	variable	variable
Z80	little	little	variable	

Appendix B

Statistical analysis material

B.1 Pairwise model comparison

Each model in each evaluation strategy is compared to each other using a paried t-test, testing for whether there are statistically significant differences in model performance. The results are shown in the tables below.

B.1.1 Endianness

Pairwise Model Comparisons KFold-all - endianness

	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
Simple1d	---	Δ = 0.0046 p = 0.0000*** (Simple1d-E > Simple1d)	Δ = 0.0003 p = 0.4984 (Simple2d > Simple1d)	Δ = 0.0050 p = 0.0000*** (Simple2d-E > Simple1d)	Δ = 0.0032 p = 0.0000*** (ResNet50 < Simple1d)	Δ = 0.0045 p = 0.0000*** (ResNet50-E > Simple1d)
Simple1d-E	Δ = 0.0046 p = 0.0000*** (Simple1d < Simple1d-E)	---	Δ = 0.0044 p = 0.0000*** (Simple2d < Simple1d-E)	Δ = 0.0003 p = 0.1120 (Simple2d-E > Simple1d-E)	Δ = 0.0078 p = 0.0000*** (ResNet50 < Simple1d-E)	Δ = 0.0001 p = 0.4850 (ResNet50-E < Simple1d-E)
Simple2d	Δ = 0.0003 p = 0.4984 (Simple1d < Simple2d)	Δ = 0.0044 p = 0.0000*** (Simple1d-E > Simple2d)	---	Δ = 0.0047 p = 0.0000*** (Simple2d-E > Simple2d)	Δ = 0.0035 p = 0.0005*** (ResNet50 < Simple2d)	Δ = 0.0042 p = 0.0000*** (ResNet50-E > Simple2d)
Simple2d-E	Δ = 0.0050 p = 0.0000*** (Simple1d < Simple2d-E)	Δ = 0.0003 p = 0.1120 (Simple1d-E < Simple2d-E)	Δ = 0.0047 p = 0.0000*** (Simple2d < Simple2d-E)	---	Δ = 0.0082 p = 0.0000*** (ResNet50 < Simple2d-E)	Δ = 0.0005 p = 0.0004*** (ResNet50-E < Simple2d-E)
ResNet50	Δ = 0.0032 p = 0.0000*** (Simple1d > ResNet50)	Δ = 0.0078 p = 0.0000*** (Simple1d-E > ResNet50)	Δ = 0.0035 p = 0.0005*** (Simple2d > ResNet50)	Δ = 0.0082 p = 0.0000*** (Simple2d-E > ResNet50)	---	Δ = 0.0077 p = 0.0000*** (ResNet50-E > ResNet50)
ResNet50-E	Δ = 0.0045 p = 0.0000*** (Simple1d < ResNet50-E)	Δ = 0.0001 p = 0.4850 (Simple1d-E > ResNet50-E)	Δ = 0.0042 p = 0.0000*** (Simple2d < ResNet50-E)	Δ = 0.0005 p = 0.0004*** (Simple2d-E > ResNet50-E)	Δ = 0.0077 p = 0.0000*** (ResNet50 < ResNet50-E)	---

Notes:
- Δ: Average accuracy difference between models
- H_0: There is no significant difference in average accuracy between the models.
- *: p < 0.05, **: p < 0.01, ***: p < 0.001
- Direction shows which model performs better
- Green cells indicate the model in the column is significantly better than the model in the row
- Red cells indicate the model in the row is significantly better than the model in the column

Figure B.1: Significance of compared model performance on the kfold endianness evaluation. The p-value refers to the probability, given that there is no significant difference between the two models, that we observe the difference that we have found.

Pairwise Model Comparisons logo - endianness

	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
Simple1d	---	Δ = 0.2094 p = 0.0000*** (Simple1d-E > Simple1d)	Δ = 0.0955 p = 0.0003*** (Simple2d < Simple1d)	Δ = 0.1616 p = 0.0000*** (Simple2d-E > Simple1d)	Δ = 0.0793 p = 0.0004*** (ResNet50 < Simple1d)	Δ = 0.0715 p = 0.0037** (ResNet50-E > Simple1d)
Simple1d-E	Δ = 0.2094 p = 0.0000*** (Simple1d < Simple1d-E)	---	Δ = 0.3049 p = 0.0000*** (Simple2d < Simple1d-E)	Δ = 0.0477 p = 0.0228* (Simple2d-E < Simple1d-E)	Δ = 0.2887 p = 0.0000*** (ResNet50 < Simple1d-E)	Δ = 0.1379 p = 0.0000*** (ResNet50-E < Simple1d-E)
Simple2d	Δ = 0.0955 p = 0.0003*** (Simple1d > Simple2d)	Δ = 0.3049 p = 0.0000*** (Simple1d-E > Simple2d)	---	Δ = 0.2572 p = 0.0000*** (Simple2d-E > Simple2d)	Δ = 0.0162 p = 0.1214 (ResNet50 > Simple2d)	Δ = 0.1670 p = 0.0000*** (ResNet50-E > Simple2d)
Simple2d-E	Δ = 0.1616 p = 0.0000*** (Simple1d < Simple2d-E)	Δ = 0.0477 p = 0.0228* (Simple1d-E > Simple2d-E)	Δ = 0.2572 p = 0.0000*** (Simple2d < Simple2d-E)	---	Δ = 0.2409 p = 0.0000*** (ResNet50 < Simple2d-E)	Δ = 0.0902 p = 0.0002*** (ResNet50-E < Simple2d-E)
ResNet50	Δ = 0.0793 p = 0.0004*** (Simple1d > ResNet50)	Δ = 0.2887 p = 0.0000*** (Simple1d-E > ResNet50)	Δ = 0.0162 p = 0.1214 (Simple2d < ResNet50)	Δ = 0.2409 p = 0.0000*** (Simple2d-E > ResNet50)	---	Δ = 0.1508 p = 0.0000*** (ResNet50-E > ResNet50)
ResNet50-E	Δ = 0.0715 p = 0.0037** (Simple1d < ResNet50-E)	Δ = 0.1379 p = 0.0000*** (Simple1d-E > ResNet50-E)	Δ = 0.1670 p = 0.0000*** (Simple2d < ResNet50-E)	Δ = 0.0902 p = 0.0002*** (Simple2d-E > ResNet50-E)	Δ = 0.1508 p = 0.0000*** (ResNet50 < ResNet50-E)	---

Notes:
- Δ: Average accuracy difference between models
- H_0: There is no significant difference in average accuracy between the models.
- *: p < 0.05, **: p < 0.01, ***: p < 0.001
- Direction shows which model performs better
- Green cells indicate the model in the column is significantly better than the model in the row
- Red cells indicate the model in the row is significantly better than the model in the column

Figure B.2: Significance of compared model performance on the Leave-One-Group-Out Cross-Validation (LOGO CV) endianness evaluation. The p-value refers to the probability, given that there is no significant difference between the two models, that we observe the difference that we have found.

Pairwise Model Comparisons BuildCross - endianness

	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
Simple1d	---	Δ = 0.1704 p = 0.0000*** (Simple1d-E < Simple1d)	Δ = 0.0233 p = 0.0000*** (Simple2d < Simple1d)	Δ = 0.1418 p = 0.0000*** (Simple2d-E < Simple1d)	Δ = 0.1025 p = 0.0000*** (ResNet50 < Simple1d)	Δ = 0.1420 p = 0.0000*** (ResNet50-E < Simple1d)
Simple1d-E	Δ = 0.1704 p = 0.0000*** (Simple1d > Simple1d-E)	---	Δ = 0.1470 p = 0.0000*** (Simple2d > Simple1d-E)	Δ = 0.0285 p = 0.0024** (Simple2d-E > Simple1d-E)	Δ = 0.0679 p = 0.0000*** (ResNet50 > Simple1d-E)	Δ = 0.0283 p = 0.0045** (ResNet50-E > Simple1d-E)
Simple2d	Δ = 0.0233 p = 0.0000*** (Simple1d > Simple2d)	Δ = 0.1470 p = 0.0000*** (Simple1d-E < Simple2d)	---	Δ = 0.1185 p = 0.0000*** (Simple2d-E < Simple2d)	Δ = 0.0792 p = 0.0000*** (ResNet50 < Simple2d)	Δ = 0.1187 p = 0.0000*** (ResNet50-E < Simple2d)
Simple2d-E	Δ = 0.1418 p = 0.0000*** (Simple1d > Simple2d-E)	Δ = 0.0285 p = 0.0024** (Simple1d-E < Simple2d-E)	Δ = 0.1185 p = 0.0000*** (Simple2d > Simple2d-E)	---	Δ = 0.0393 p = 0.0013** (ResNet50 > Simple2d-E)	Δ = 0.0002 p = 0.9801 (ResNet50-E < Simple2d-E)
ResNet50	Δ = 0.1025 p = 0.0000*** (Simple1d > ResNet50)	Δ = 0.0679 p = 0.0000*** (Simple1d-E < ResNet50)	Δ = 0.0792 p = 0.0000*** (Simple2d > ResNet50)	Δ = 0.0393 p = 0.0013** (Simple2d-E < ResNet50)	---	Δ = 0.0395 p = 0.0010*** (ResNet50-E < ResNet50)
ResNet50-E	Δ = 0.1420 p = 0.0000*** (Simple1d > ResNet50-E)	Δ = 0.0283 p = 0.0045** (Simple1d-E < ResNet50-E)	Δ = 0.1187 p = 0.0000*** (Simple2d > ResNet50-E)	Δ = 0.0002 p = 0.9801 (Simple2d-E > ResNet50-E)	Δ = 0.0395 p = 0.0010*** (ResNet50 > ResNet50-E)	---

Notes:
- Δ: Average accuracy difference between models
- H_0: There is no significant difference in average accuracy between the models.
- *: p < 0.05, **: p < 0.01, ***: p < 0.001
- Direction shows which model performs better
- Green cells indicate the model in the column is significantly better than the model in the row
- Red cells indicate the model in the row is significantly better than the model in the column

Figure B.3: Significance of compared model performance on the ISAdetect-BuildCross evaluation. The p-value refers to the probability, given that there is no significant difference between the two models, that we observe the difference that we have found.

Pairwise Model Comparisons CpuRec - endianness

	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
Simple1d	---	Δ = 0.0375 p = 0.0014** (Simple1d-E > Simple1d)	Δ = 0.0179 p = 0.0125* (Simple2d < Simple1d)	Δ = 0.0464 p = 0.0034** (Simple2d-E > Simple1d)	Δ = 0.0705 p = 0.0000*** (ResNet50 < Simple1d)	Δ = 0.0473 p = 0.0001*** (ResNet50-E > Simple1d)
Simple1d-E	Δ = 0.0375 p = 0.0014** (Simple1d < Simple1d-E)	---	Δ = 0.0554 p = 0.0000*** (Simple2d < Simple1d-E)	Δ = 0.0089 p = 0.3867 (Simple2d-E > Simple1d-E)	Δ = 0.1080 p = 0.0000*** (ResNet50 < Simple1d-E)	Δ = 0.0098 p = 0.3379 (ResNet50-E > Simple1d-E)
Simple2d	Δ = 0.0179 p = 0.0125* (Simple1d > Simple2d)	Δ = 0.0554 p = 0.0000*** (Simple1d-E > Simple2d)	---	Δ = 0.0643 p = 0.0000*** (Simple2d-E > Simple2d)	Δ = 0.0527 p = 0.0000*** (ResNet50 < Simple2d)	Δ = 0.0652 p = 0.0000*** (ResNet50-E > Simple2d)
Simple2d-E	Δ = 0.0464 p = 0.0034** (Simple1d < Simple2d-E)	Δ = 0.0089 p = 0.3867 (Simple1d-E < Simple2d-E)	Δ = 0.0643 p = 0.0000*** (Simple2d < Simple2d-E)	---	Δ = 0.1170 p = 0.0000*** (ResNet50 < Simple2d-E)	Δ = 0.0009 p = 0.9512 (ResNet50-E > Simple2d-E)
ResNet50	Δ = 0.0705 p = 0.0000*** (Simple1d > ResNet50)	Δ = 0.1080 p = 0.0000*** (Simple1d-E > ResNet50)	Δ = 0.0527 p = 0.0000*** (Simple2d > ResNet50)	Δ = 0.1170 p = 0.0000*** (Simple2d-E > ResNet50)	---	Δ = 0.1179 p = 0.0000*** (ResNet50-E > ResNet50)
ResNet50-E	Δ = 0.0473 p = 0.0001*** (Simple1d < ResNet50-E)	Δ = 0.0098 p = 0.3379 (Simple1d-E < ResNet50-E)	Δ = 0.0652 p = 0.0000*** (Simple2d < ResNet50-E)	Δ = 0.0009 p = 0.9512 (Simple2d-E < ResNet50-E)	Δ = 0.1179 p = 0.0000*** (ResNet50 < ResNet50-E)	---

Notes:
- Δ: Average accuracy difference between models
- H_0: There is no significant difference in average accuracy between the models.
- *: p < 0.05, **: p < 0.01, ***: p < 0.001
- Direction shows which model performs better
- Green cells indicate the model in the column is significantly better than the model in the row
- Red cells indicate the model in the row is significantly better than the model in the column

Figure B.4: Significance of compared model performance on the ISAdetect-CpuRec evaluation. The p-value refers to the probability, given that there is no significant difference between the two models, that we observe the difference that we have found.

Pairwise Model Comparisons Combined - endianness

	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
Simple1d	---	Δ = 0.1607 p = 0.0000*** (Simple1d-E > Simple1d)	Δ = 0.0491 p = 0.0000*** (Simple2d > Simple1d)	Δ = 0.1491 p = 0.0000*** (Simple2d-E > Simple1d)	Δ = 0.0679 p = 0.0000*** (ResNet50 > Simple1d)	Δ = 0.1705 p = 0.0000*** (ResNet50-E > Simple1d)
Simple1d-E	Δ = 0.1607 p = 0.0000*** (Simple1d < Simple1d-E)	---	Δ = 0.1116 p = 0.0000*** (Simple2d < Simple1d-E)	Δ = 0.0116 p = 0.1256 (Simple2d-E < Simple1d-E)	Δ = 0.0929 p = 0.0000*** (ResNet50 < Simple1d-E)	Δ = 0.0098 p = 0.2698 (ResNet50-E > Simple1d-E)
Simple2d	Δ = 0.0491 p = 0.0000*** (Simple1d < Simple2d)	Δ = 0.1116 p = 0.0000*** (Simple1d-E > Simple2d)	---	Δ = 0.1000 p = 0.0000*** (Simple2d-E > Simple2d)	Δ = 0.0187 p = 0.2030 (ResNet50 > Simple2d)	Δ = 0.1214 p = 0.0000*** (ResNet50-E > Simple2d)
Simple2d-E	Δ = 0.1491 p = 0.0000*** (Simple1d < Simple2d-E)	Δ = 0.0116 p = 0.1256 (Simple1d-E > Simple2d-E)	Δ = 0.1000 p = 0.0000*** (Simple2d < Simple2d-E)	---	Δ = 0.0813 p = 0.0000*** (ResNet50 < Simple2d-E)	Δ = 0.0214 p = 0.0453* (ResNet50-E > Simple2d-E)
ResNet50	Δ = 0.0679 p = 0.0000*** (Simple1d < ResNet50)	Δ = 0.0929 p = 0.0000*** (Simple1d-E > ResNet50)	Δ = 0.0187 p = 0.2030 (Simple2d < ResNet50)	Δ = 0.0813 p = 0.0000*** (Simple2d-E > ResNet50)	---	Δ = 0.1027 p = 0.0000*** (ResNet50-E > ResNet50)
ResNet50-E	Δ = 0.1705 p = 0.0000*** (Simple1d < ResNet50-E)	Δ = 0.0098 p = 0.2698 (Simple1d-E < ResNet50-E)	Δ = 0.1214 p = 0.0000*** (Simple2d < ResNet50-E)	Δ = 0.0214 p = 0.0453* (Simple2d-E < ResNet50-E)	Δ = 0.1027 p = 0.0000*** (ResNet50 < ResNet50-E)	---

Notes:
- Δ: Average accuracy difference between models
- H_0: There is no significant difference in average accuracy between the models.
- *: p < 0.05, **: p < 0.01, ***: p < 0.001
- Direction shows which model performs better
- Green cells indicate the model in the column is significantly better than the model in the row
- Red cells indicate the model in the row is significantly better than the model in the column

Figure B.5: Significance of compared model performance on the Combined-CpuRec evaluation. The p-value refers to the probability, given that there is no significant difference between the two models, that we observe the difference that we have found.

B.1.2 Instruction width type

Pairwise Model Comparisons KFold-all - instructionwidth_type

	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
Simple1d	---	Δ = 0.0062 p = 0.0000*** (Simple1d-E > Simple1d)	Δ = 0.0019 p = 0.0060** (Simple2d > Simple1d)	Δ = 0.0065 p = 0.0000*** (Simple2d-E > Simple1d)	Δ = 0.0018 p = 0.0119* (ResNet50 < Simple1d)	Δ = 0.0053 p = 0.0000*** (ResNet50-E > Simple1d)
Simple1d-E	Δ = 0.0062 p = 0.0000*** (Simple1d < Simple1d-E)	---	Δ = 0.0043 p = 0.0000*** (Simple2d < Simple1d-E)	Δ = 0.0002 p = 0.0428* (Simple2d-E > Simple1d-E)	Δ = 0.0080 p = 0.0000*** (ResNet50 < Simple1d-E)	Δ = 0.0009 p = 0.0000*** (ResNet50-E < Simple1d-E)
Simple2d	Δ = 0.0019 p = 0.0060** (Simple1d < Simple2d)	Δ = 0.0043 p = 0.0000*** (Simple1d-E > Simple2d)	---	Δ = 0.0046 p = 0.0000*** (Simple2d-E > Simple2d)	Δ = 0.0037 p = 0.0000*** (ResNet50 < Simple2d)	Δ = 0.0034 p = 0.0000*** (ResNet50-E > Simple2d)
Simple2d-E	Δ = 0.0065 p = 0.0000*** (Simple1d < Simple2d-E)	Δ = 0.0002 p = 0.0428* (Simple1d-E < Simple2d-E)	Δ = 0.0046 p = 0.0000*** (Simple2d < Simple2d-E)	---	Δ = 0.0082 p = 0.0000*** (ResNet50 < Simple2d-E)	Δ = 0.0012 p = 0.0000*** (ResNet50-E < Simple2d-E)
ResNet50	Δ = 0.0018 p = 0.0119* (Simple1d > ResNet50)	Δ = 0.0080 p = 0.0000*** (Simple1d-E > ResNet50)	Δ = 0.0037 p = 0.0000*** (Simple2d > ResNet50)	Δ = 0.0082 p = 0.0000*** (Simple2d-E > ResNet50)	---	Δ = 0.0071 p = 0.0000*** (ResNet50-E > ResNet50)
ResNet50-E	Δ = 0.0053 p = 0.0000*** (Simple1d < ResNet50-E)	Δ = 0.0009 p = 0.0000*** (Simple1d-E > ResNet50-E)	Δ = 0.0034 p = 0.0000*** (Simple2d < ResNet50-E)	Δ = 0.0012 p = 0.0000*** (Simple2d-E > ResNet50-E)	Δ = 0.0071 p = 0.0000*** (ResNet50 < ResNet50-E)	---

Notes:
- Δ: Average accuracy difference between models
- H_0: There is no significant difference in average accuracy between the models.
- *: p < 0.05, **: p < 0.01, ***: p < 0.001
- Direction shows which model performs better
- Green cells indicate the model in the column is significantly better than the model in the row
- Red cells indicate the model in the row is significantly better than the model in the column

Figure B.6: Significance of compared model performance on the K-fold cross validation instruction width type evaluation. The p-value refers to the probability, given that there is no significant difference between the two models, that we observe the difference that we have found.

Pairwise Model Comparisons logo - instructionwidth_type

	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
Simple1d	---	Δ = 0.1307 p = 0.0000*** (Simple1d-E > Simple1d)	Δ = 0.0153 p = 0.4812 (Simple2d < Simple1d)	Δ = 0.0935 p = 0.0001*** (Simple2d-E > Simple1d)	Δ = 0.0390 p = 0.0627 (ResNet50 > Simple1d)	Δ = 0.0937 p = 0.0001*** (ResNet50-E > Simple1d)
Simple1d-E	Δ = 0.1307 p = 0.0000*** (Simple1d < Simple1d-E)	---	Δ = 0.1461 p = 0.0003*** (Simple2d < Simple1d-E)	Δ = 0.0372 p = 0.0086** (Simple2d-E < Simple1d-E)	Δ = 0.0917 p = 0.0002*** (ResNet50 < Simple1d-E)	Δ = 0.0370 p = 0.0765 (ResNet50-E < Simple1d-E)
Simple2d	Δ = 0.0153 p = 0.4812 (Simple1d > Simple2d)	Δ = 0.1461 p = 0.0003*** (Simple1d-E > Simple2d)	---	Δ = 0.1088 p = 0.0005*** (Simple2d-E > Simple2d)	Δ = 0.0543 p = 0.0262* (ResNet50 > Simple2d)	Δ = 0.1091 p = 0.0006*** (ResNet50-E > Simple2d)
Simple2d-E	Δ = 0.0935 p = 0.0001*** (Simple1d < Simple2d-E)	Δ = 0.0372 p = 0.0086** (Simple1d-E > Simple2d-E)	Δ = 0.1088 p = 0.0005*** (Simple2d < Simple2d-E)	---	Δ = 0.0545 p = 0.0008*** (ResNet50 < Simple2d-E)	Δ = 0.0003 p = 0.9886 (ResNet50-E > Simple2d-E)
ResNet50	Δ = 0.0390 p = 0.0627 (Simple1d < ResNet50)	Δ = 0.0917 p = 0.0002*** (Simple1d-E > ResNet50)	Δ = 0.0543 p = 0.0262* (Simple2d < ResNet50)	Δ = 0.0545 p = 0.0008*** (Simple2d-E > ResNet50)	---	Δ = 0.0547 p = 0.0186* (ResNet50-E > ResNet50)
ResNet50-E	Δ = 0.0937 p = 0.0001*** (Simple1d < ResNet50-E)	Δ = 0.0370 p = 0.0765 (Simple1d-E > ResNet50-E)	Δ = 0.1091 p = 0.0006*** (Simple2d < ResNet50-E)	Δ = 0.0003 p = 0.9886 (Simple2d-E < ResNet50-E)	Δ = 0.0547 p = 0.0186* (ResNet50 < ResNet50-E)	---

Notes:
- Δ: Average accuracy difference between models
- H_0: There is no significant difference in average accuracy between the models.
- *: p < 0.05, **: p < 0.01, ***: p < 0.001
- Direction shows which model performs better
- Green cells indicate the model in the column is significantly better than the model in the row
- Red cells indicate the model in the row is significantly better than the model in the column

Figure B.7: Significance of compared model performance on the LOGO CV instruction width type evaluation. The p-value refers to the probability, given that there is no significant difference between the two models, that we observe the difference that we have found.

Pairwise Model Comparisons BuildCross - instructionwidth_type

	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
Simple1d	---	Δ = 0.0494 p = 0.0008*** (Simple1d-E < Simple1d)	Δ = 0.0620 p = 0.0000*** (Simple2d > Simple1d)	Δ = 0.0131 p = 0.3346 (Simple2d-E > Simple1d)	Δ = 0.0058 p = 0.4847 (ResNet50 > Simple1d)	Δ = 0.0671 p = 0.0000*** (ResNet50-E < Simple1d)
Simple1d-E	Δ = 0.0494 p = 0.0008*** (Simple1d > Simple1d-E)	---	Δ = 0.1113 p = 0.0000*** (Simple2d > Simple1d-E)	Δ = 0.0625 p = 0.0000*** (Simple2d-E > Simple1d-E)	Δ = 0.0551 p = 0.0006*** (ResNet50 > Simple1d-E)	Δ = 0.0178 p = 0.1616 (ResNet50-E < Simple1d-E)
Simple2d	Δ = 0.0620 p = 0.0000*** (Simple1d < Simple2d)	Δ = 0.1113 p = 0.0000*** (Simple1d-E < Simple2d)	---	Δ = 0.0489 p = 0.0020** (Simple2d-E < Simple2d)	Δ = 0.0562 p = 0.0000*** (ResNet50 < Simple2d)	Δ = 0.1291 p = 0.0000*** (ResNet50-E < Simple2d)
Simple2d-E	Δ = 0.0131 p = 0.3346 (Simple1d < Simple2d-E)	Δ = 0.0625 p = 0.0000*** (Simple1d-E < Simple2d-E)	Δ = 0.0489 p = 0.0020** (Simple2d > Simple2d-E)	---	Δ = 0.0073 p = 0.6048 (ResNet50 < Simple2d-E)	Δ = 0.0802 p = 0.0000*** (ResNet50-E < Simple2d-E)
ResNet50	Δ = 0.0058 p = 0.4847 (Simple1d < ResNet50)	Δ = 0.0551 p = 0.0006*** (Simple1d-E < ResNet50)	Δ = 0.0562 p = 0.0000*** (Simple2d > ResNet50)	Δ = 0.0073 p = 0.6048 (Simple2d-E > ResNet50)	---	Δ = 0.0729 p = 0.0000*** (ResNet50-E < ResNet50)
ResNet50-E	Δ = 0.0671 p = 0.0000*** (Simple1d > ResNet50-E)	Δ = 0.0178 p = 0.1616 (Simple1d-E > ResNet50-E)	Δ = 0.1291 p = 0.0000*** (Simple2d > ResNet50-E)	Δ = 0.0802 p = 0.0000*** (Simple2d-E > ResNet50-E)	Δ = 0.0729 p = 0.0000*** (ResNet50 > ResNet50-E)	---

Notes:
- Δ: Average accuracy difference between models
- H_0: There is no significant difference in average accuracy between the models.
- *: p < 0.05, **: p < 0.01, ***: p < 0.001
- Direction shows which model performs better
- Green cells indicate the model in the column is significantly better than the model in the row
- Red cells indicate the model in the row is significantly better than the model in the column

Figure B.8: Significance of compared model performance on the ISAdetect-BuildCross instruction width type evaluation. The p-value refers to the probability, given that there is no significant difference between the two models, that we observe the difference that we have found.

Pairwise Model Comparisons CpuRec - instructionwidth_type

	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
Simple1d	---	Δ = 0.0048 p = 0.8026 (Simple1d-E < Simple1d)	Δ = 0.0651 p = 0.0001*** (Simple2d > Simple1d)	Δ = 0.0240 p = 0.1973 (Simple2d-E > Simple1d)	Δ = 0.0295 p = 0.1319 (ResNet50 > Simple1d)	Δ = 0.0055 p = 0.7543 (ResNet50-E < Simple1d)
Simple1d-E	Δ = 0.0048 p = 0.8026 (Simple1d > Simple1d-E)	---	Δ = 0.0699 p = 0.0000*** (Simple2d > Simple1d-E)	Δ = 0.0288 p = 0.0052** (Simple2d-E > Simple1d-E)	Δ = 0.0342 p = 0.0067** (ResNet50 > Simple1d-E)	Δ = 0.0007 p = 0.9430 (ResNet50-E < Simple1d-E)
Simple2d	Δ = 0.0651 p = 0.0001*** (Simple1d < Simple2d)	Δ = 0.0699 p = 0.0000*** (Simple1d-E < Simple2d)	---	Δ = 0.0411 p = 0.0028** (Simple2d-E < Simple2d)	Δ = 0.0356 p = 0.0105* (ResNet50 < Simple2d)	Δ = 0.0705 p = 0.0000*** (ResNet50-E < Simple2d)
Simple2d-E	Δ = 0.0240 p = 0.1973 (Simple1d < Simple2d-E)	Δ = 0.0288 p = 0.0052** (Simple1d-E < Simple2d-E)	Δ = 0.0411 p = 0.0028** (Simple2d > Simple2d-E)	---	Δ = 0.0055 p = 0.5957 (ResNet50 > Simple2d-E)	Δ = 0.0295 p = 0.0034** (ResNet50-E < Simple2d-E)
ResNet50	Δ = 0.0295 p = 0.1319 (Simple1d < ResNet50)	Δ = 0.0342 p = 0.0067** (Simple1d-E < ResNet50)	Δ = 0.0356 p = 0.0105* (Simple2d > ResNet50)	Δ = 0.0055 p = 0.5957 (Simple2d-E < ResNet50)	---	Δ = 0.0349 p = 0.0009*** (ResNet50-E < ResNet50)
ResNet50-E	Δ = 0.0055 p = 0.7543 (Simple1d > ResNet50-E)	Δ = 0.0007 p = 0.9430 (Simple1d-E > ResNet50-E)	Δ = 0.0705 p = 0.0000*** (Simple2d > ResNet50-E)	Δ = 0.0295 p = 0.0034** (Simple2d-E > ResNet50-E)	Δ = 0.0349 p = 0.0009*** (ResNet50 > ResNet50-E)	---

Notes:
- Δ: Average accuracy difference between models
- H_0: There is no significant difference in average accuracy between the models.
- *: p < 0.05, **: p < 0.01, ***: p < 0.001
- Direction shows which model performs better
- Green cells indicate the model in the column is significantly better than the model in the row
- Red cells indicate the model in the row is significantly better than the model in the column

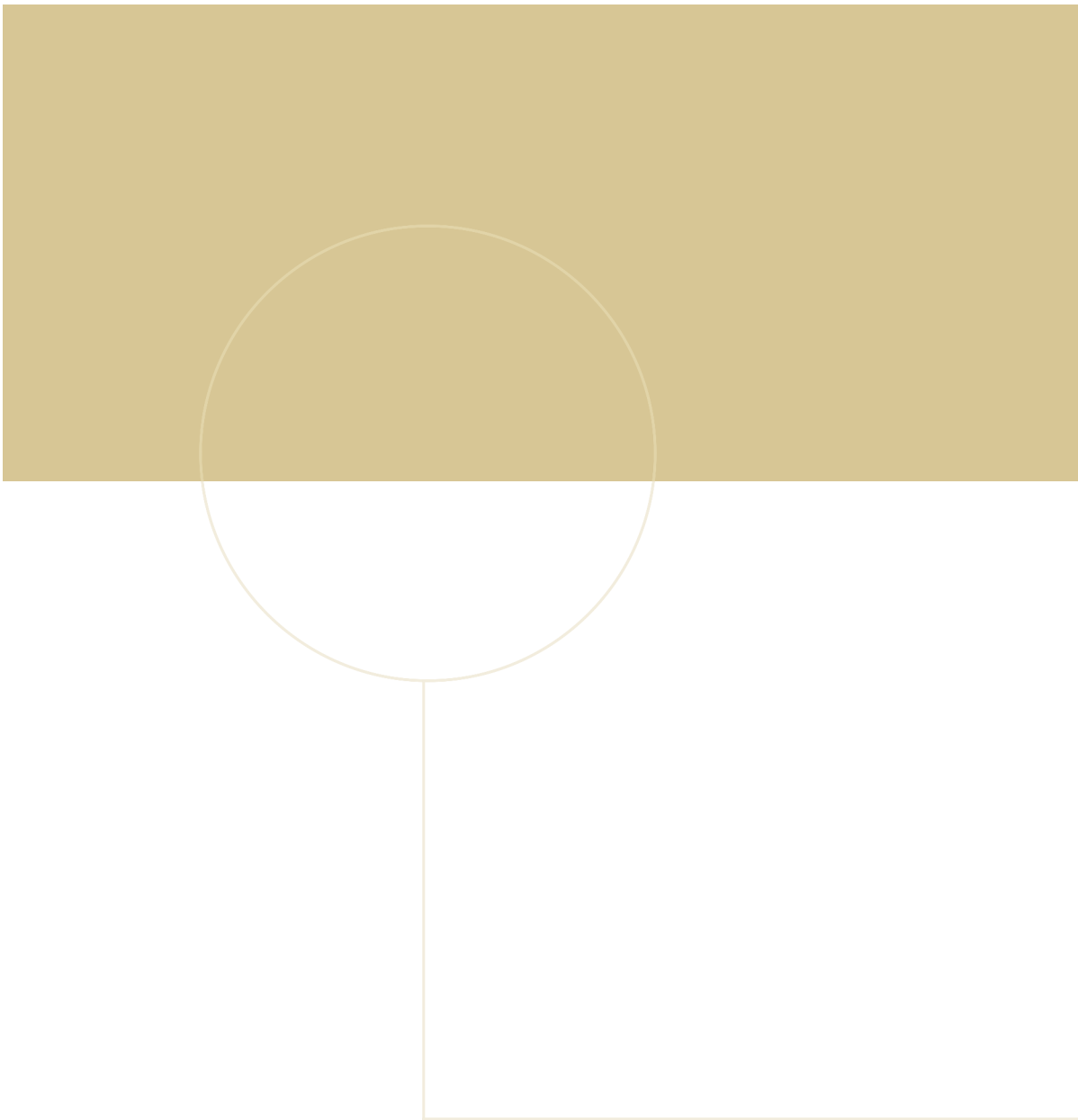
Figure B.9: Significance of compared model performance on the ISAdetect-CpuRec instruction width type evaluation. The p-value refers to the probability, given that there is no significant difference between the two models, that we observe the difference that we have found.

Pairwise Model Comparisons Combined - instructionwidth_type

	Simple1d	Simple1d-E	Simple2d	Simple2d-E	ResNet50	ResNet50-E
Simple1d	---	Δ = 0.0514 p = 0.0000*** (Simple1d-E > Simple1d)	Δ = 0.0110 p = 0.5357 (Simple2d < Simple1d)	Δ = 0.0863 p = 0.0000*** (Simple2d-E > Simple1d)	Δ = 0.0103 p = 0.3845 (ResNet50 < Simple1d)	Δ = 0.0630 p = 0.0000*** (ResNet50-E > Simple1d)
Simple1d-E	Δ = 0.0514 p = 0.0000*** (Simple1d < Simple1d-E)	---	Δ = 0.0623 p = 0.0009*** (Simple2d < Simple1d-E)	Δ = 0.0349 p = 0.0003*** (Simple2d-E > Simple1d-E)	Δ = 0.0616 p = 0.0000*** (ResNet50 < Simple1d-E)	Δ = 0.0116 p = 0.1871 (ResNet50-E > Simple1d-E)
Simple2d	Δ = 0.0110 p = 0.5357 (Simple1d > Simple2d)	Δ = 0.0623 p = 0.0009*** (Simple1d-E > Simple2d)	---	Δ = 0.0973 p = 0.0000*** (Simple2d-E > Simple2d)	Δ = 0.0007 p = 0.9682 (ResNet50 > Simple2d)	Δ = 0.0740 p = 0.0002*** (ResNet50-E > Simple2d)
Simple2d-E	Δ = 0.0863 p = 0.0000*** (Simple1d < Simple2d-E)	Δ = 0.0349 p = 0.0003*** (Simple1d-E < Simple2d-E)	Δ = 0.0973 p = 0.0000*** (Simple2d < Simple2d-E)	---	Δ = 0.0966 p = 0.0000*** (ResNet50 < Simple2d-E)	Δ = 0.0233 p = 0.0212* (ResNet50-E < Simple2d-E)
ResNet50	Δ = 0.0103 p = 0.3845 (Simple1d > ResNet50)	Δ = 0.0616 p = 0.0000*** (Simple1d-E > ResNet50)	Δ = 0.0007 p = 0.9682 (Simple2d < ResNet50)	Δ = 0.0966 p = 0.0000*** (Simple2d-E > ResNet50)	---	Δ = 0.0733 p = 0.0000*** (ResNet50-E > ResNet50)
ResNet50-E	Δ = 0.0630 p = 0.0000*** (Simple1d < ResNet50-E)	Δ = 0.0116 p = 0.1871 (Simple1d-E < ResNet50-E)	Δ = 0.0740 p = 0.0002*** (Simple2d < ResNet50-E)	Δ = 0.0233 p = 0.0212* (Simple2d-E > ResNet50-E)	Δ = 0.0733 p = 0.0000*** (ResNet50 < ResNet50-E)	---

Notes:
- Δ: Average accuracy difference between models
- H_0: There is no significant difference in average accuracy between the models.
- *: p < 0.05, **: p < 0.01, ***: p < 0.001
- Direction shows which model performs better
- Green cells indicate the model in the column is significantly better than the model in the row
- Red cells indicate the model in the row is significantly better than the model in the column

Figure B.10: Significance of compared model performance on the Combined-CpuRec instruction width type evaluation. The p-value refers to the probability, given that there is no significant difference between the two models, that we observe the difference that we have found.



NTNU

Norwegian University of
Science and Technology