Alma Mater Studiorum - Università di Bologna

DOTTORATO DI RICERCA IN

INGEGNERIA ELETTRONICA, TELECOMUNICAZIONI E TECNOLOGIE DELL'INFORMAZIONE

Ciclo 36

Settore Concorsuale: 09/H1 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Settore Scientifico Disciplinare: ING-INF/05 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

ADDRESSING CHALLENGES IN HETEROGENEOUS EMBEDDED SYSTEMS PROGRAMMING: SECURITY AND PERFORMANCE

Presentata da: Emanuele Parisi

Coordinatore Dottorato

Aldo Romani

Supervisore

Andrea Acquaviva

Co-supervisore

Luca Benini

Esame finale anno 2024

Abstract

Embedded systems are increasingly integral to daily life, improving and facilitating the efficiency of an increasing number of systems, spanning from industrial plants, to healthcare devices and autonomous vehicles. The emergence of Cyber-Physical Systems has enhanced the digital computational capabilities of modern Embedded Systems with access to real-world data coming from sensor readings, and the possibility to act in the physical world through the control of actuators. As the architecture of these systems becomes increasingly complex, incorporating parallel clusters and custom hardware accelerators, as well as being integrated into safety-critical scenarios, utilizing their computational capabilities effectively poses numerous challenges. Additionally, ensuring security features are in place to avoid harm to individuals and assets is a crucial concern. The research primarily addresses challenges in contemporary Embedded Systems, focusing on platform optimization and security enforcement.

The initial section of this study delves into the application of machine learning methods to efficiently determine the optimal number of cores for a parallel RISC-V cluster to minimize energy consumption. This is achieved through static source code analysis. Results from experiments conducted on a contemporary RISC-V System-on-Chip demonstrate that automated platform configuration is not only viable but also that there is a moderate performance trade-off when relying solely on static features. Consequently, this preliminary result shows room for exploration and development of more intricate platform configuration systems based on static source code analysis techniques.

The second part focuses on addressing the problem of heterogeneous device mapping, which involves assigning tasks to the most suitable computational device in a heterogeneous platform for optimal runtime. The contribution of this section lies in optimizing the training framework for deep-learning models aimed at solving this problem. A new data pre-processing technique is introduced, along with a training framework called Siamese Networks, that enhances the classification performance of DeepLLVM, an advanced approach for task mapping. Importantly, these proposed approaches not only improve the classification performance of DeepLLVM but are also independent from the specific deep-learning model used. This offers potential enhancements to accuracy across various state-of-the-art methodologies.

The final aspect of this research work focuses on addressing issues concerning the binary exploitation of software running in modern Embedded Systems. It proposes an architecture to implement Control-Flow Integrity in embedded platforms with a Root-of-Trust, aiming to enhance security guarantees without extensive hardware modifications. The approach involves enhancing the architecture of a modern RISC-V platform for autonomous vehicles by implementing a side-channel communication mechanism that relays control-flow changes executed by the process running on the host core to the Root-of-Trust. The firmware of the Root-of-Trust is then modified to analyze and enforce the desired Control-Flow Integrity policy in software, notifying any security violations back to the host core. This approach has minimal impact on system performance as shown through testing using well-known benchmark suites, demonstrating its feasibility and effectiveness in enhancing security within embedded RISC-V platforms.

Contents

List of Figures

List of Tables

Introduction 1

Embedded systems are becoming increasingly common in many domains, they have become an integral part of our daily lives, enhancing it in every way possible. From consumer electronics to industrial automation, the advances in hardware architectures and integration in complex systems along side with sensors and actuators have paved the way for creation of Cyber-Physical Systems. These systems are vital for making everyday tasks more efficient and effective. The importance of Cyber-Physical Systems has grown in recent years, due to their ability to implement complex monitoring and control tasks across a variety of industries including manufacturing, energy, healthcare, and transportation. Such pervasive adoption, the need to run increasingly complex applications efficiently, and their involvement in safety-critical environments pose numerous challenges in their programming and security enforcement on such systems.

1.1 Challenges in system performance

As Dennard's scaling approaches its limits and the progress of Moore's law slows, contemporary embedded architectures strive to achieve energy efficiency and high computational performance by integrating general-purpose CPUs with hardware accelerators. They also utilize parallel clusters, such as CPU-GPU or CPU-FPGA heterogeneous architectures, to leverage the strengths of different processing units and optimize overall performance. In addition, platform configuration plays a vital role in this context by utilizing scalable parallelism, optimized memory access patterns, and flexible low power states to enhance energy efficiency. Considering the computational demands, determining the optimal configuration of cores for energy efficiency depends on factors such as processing pressure and power consumption across different functional states. Embedded systems have embraced heterogeneity and configurability; however, selecting the most appropriate accelerator and its setup for specific kernels remains a challenge that requires expertise in this domain. Although there have been attempts to automate hardware selection and configuration using machine learning methods, numerous challenges still persist in this area. One important consideration is determining the appropriate level of abstraction for representing a program, whether it should be in a high-level programming language, compiler intermediate representation, or assembly. Additionally, there are ongoing research efforts focused on defining transformations to encode software into formats that can be analyzed by machine learning models. Another challenging aspect is developing strategies to effectively train these models when faced with limited samples available in modern datasets.

1.2 Challenges in system security

Embedded systems are commonly used in critical infrastructure and applications, including healthcare implantable devices, energy facilities, and automotive systems. It is crucial to address security vulnerabilities in these systems as they can have severe consequences. For instance, security issues in smart grids can result in service disruptions and economic losses. Similarly, information leaks or data tampering from medical implantable devices and automotive control systems can compromise their functionality and pose significant risks to people's lives. An additional complication is that deeply embedded systems are frequently programmed in languages that lack memory safety, leading to a higher likelihood of security vulnerabilities due to software bugs during development. Although secure boot and firmware verification protocols authenticate the software at startup, these measures cannot prevent a determined attacker from exploiting any read or write vulnerability in the code. This allows them to bypass traditional memory protection mechanisms and manipulate the control flow of the targeted program using "Code-Reuse Attack" techniques with malicious intent. To address these security risks, present-day System-on-Chip designs include Control-Flow Integrity enforcement policies. These policies aim to ensure that the control flow of an application follows the predetermined constraints established by the developer during design and alert the platform runtime if any violations occur. However, implementing a flexible CFI scheme on modern embedded systems poses challenges such as avoiding substantial area overheads or developing a hardware modules from scratch while still allowing for potential updates after deployment. Embedded systems in the modern era are faced with the challenge of being connected to public, unsafe networks. This connectivity raises concerns about the security of these devices. Although there are existing security measures to ensure data confidentiality, integrity, and authenticity between embedded systems, it is still necessary to provide user space applications with access to secure enclaves for storing cryptographic secrets in tamper-proof memories that can only be accessed by trusted software. Additionally, incorporating cryptographic accelerators can enhance the execution speed of essential security operations including digest computation, encryption of data streams, and signature verification.

1.3 Outline

This thesis aims to contribute to the advancements in techniques for enhancing the efficiency and security of modern embedded systems. The rest of this manuscript is divided into two parts, which discuss various techniques that address challenges related to programming and security enforcement in these systems. The first part focuses on analyzing source code using machine and deep-learning models to improve energy efficiency and runtime performance in modern embedded systems. The background section provides an overview of state-of-the-art techniques for representing source code and extracting features. In the second chapter, we address the challenge of finding the best configuration for a parallel embedded RISC-V cluster through traditional machine learning methods. Additionally, we evaluate how static analysis compares to dynamic feature extraction when it comes to achieving accurate classifications. The third chapter introduces the problem of heterogeneous device mapping, a binary classification problem when a deep neural network has to learn whether a task is optimally mapped in a CPU or on a hardware accelerators. It also describes how CNN can be used for analyze token streams and compares the accuracy with recurrent models. Finally, chapter four closes the first part, describing a set of techniques based on graph neural networks auto-encoders to learn a language model for source code. The second part of the thesis focuses on addressing security concerns in modern embedded systems, specifically ensuring execution authenticity and improving network stream performance through the utilization of hardware accelerators on RISC-V platforms. The background chapter presents a detailed overview of the reference RISC-V platform used for developing our proposed solution. It also provides an explanation of the architecture employed by OpenTitan, our chosen root-of-trust mechanism for enforcing security measures, along with an exploration into Code Reuse Attacks and Control-Flow Integrity principles. In Chapter 2, we propose a solution to enforce arbitrary Control-Flow Integrity policies within OpenTitan's Root-of-Trust infrastructure while comparing this approach against alternative enforcement schemes that rely on custom hardware accelerators in terms of their impact on execution and system resources. Chapter 3 showcases how the OpenTitan Root-of-Trust can seamlessly accelerate cryptographic operations compared to software-based execution methods. The final chapter of the thesis outlines the key findings and proposes future directions for addressing programming and security issues in latest embedded systems. It summarizes the main results obtained and suggests potential focus areas to overcome the challenges identified.

Background 2

2.1 Techniques for source code analysis

Computational problems are solved using algorithms, which consist of a series of operations that provide a solution based on a given input data. In the context of computer programming, these algorithms are expressed in programming languages following specific rules and procedures. However, this type of representation may not be suitable for machine learning models that require numerical features as input. To overcome this challenge, researchers in the field of AI-based source code analysis have proposed various techniques to transform code into numerical representations compatible with machine learning models. The following subsections describe state-of-the-art methodologies for representing source code as numerical input. Subsection 2.1.1 discusses the level of abstraction at which source code can be analyzed. Subsection 2.1.2 focuses on structural representation of sources, and it discusses two ways of observing code: as a sequence of tokens, or as a graph. Finally, Subsection 2.1.3 details how to translate a program into a numerical representation.

2.1.1 Source code language abstraction

Previous studies in this area have investigated various approaches to represent source code before utilizing it in machine learning models [2]. These approaches typically involve either using the original representation in a high-level programming language or exploiting intermediate representations generated by compilers. Utilizing the original high-level language representation offers several advantages, including independence from specific compilers, platforms, and optimization challenges. It also provides flexibility for developing different heuristics. Preprocessing steps can be applied initially to remove semantically irrelevant elements such as comments. Furthermore, this approach preserves the expressive nature of high-level code structures while ensuring robustness in classification tasks.

Employing intermediate representations generated by compilers, such as LLVM Intermediate Representation (LLVM-IR) [37] or GCC Gimple [40], has emerged as

C/C++

```c
void foo(float *v, unsigned n) {
    for (unsigned i=0; i<n; i++)
        v[i] = sqrt(v[i]) + 1;
}
```

LLVM-IR

```llvm
define void @foo(float*, unsigned int)(ptr %0, i32 %1) {
    %3 = icmp eq i32 %1, 0
    br i1 %3, label %4, label %5
4:
    ret void
5:
    %6 = phi i32 [ %11, %5 ], [ 0, %2 ]
    %7 = getelementptr inbounds float, ptr %0, i32 %6
    %8 = load float, ptr %7, align 4
    %9 = tail call float @sqrtf(float noundef %8)
    %10 = fadd float %9, 1.000000e+00
    store float %10, ptr %7, align 4
    %11 = add nuw i32 %6, 1
    %12 = icmp eq i32 %11, %1
    br i1 %12, label %4, label %5
}
```

Figure 2.1: Comparison between a linear algebra function expressed in C or in LLVM-IR.

a preferable alternative [4]. This approach has several advantages as well. Firstly, intermediate code already undergoes preliminary optimization through high-level transformations. Moreover, it boosts enhanced hardware independence compared to machine-specific assembly code and it serves as a versatile representation that can be utilized across different high-level languages. Its flexibility allows for complex compilation decisions including efficient allocation of code fragments to architecture devices [5]. Furthermore, LLVM-IR enables the application of machine learning techniques in order to gain insights into the characteristics of these code fragments by transforming these properties into fixed-length feature vectors suitable for standard machine learning algorithms. However, their further optimization remains crucial in this context and the primary objective of this manuscript is to enhance the field of knowledge representation by utilizing LLVM-IR as a foundational basis. The subsequent sections offer introductions to the syntax and distinctive characteristics of LLVM-IR.

In the field of compiler technology, LLVM-IR is widely recognized for its adaptability and often compared to an assembly-like representation. Despite its resemblance to low-level assembly code, as shown in Figure 2.1, LLVM-IR remains independent of specific target Instruction Set Architectures, focusing instead on its core properties such as platform independence, language neutrality, efficiency, modularity, support for control and data flow analysis. This foundation makes it instrumental in enabling various compiler optimizations and analyses. As a result of these attributes, LLVM-IR holds significant relevance both in academic research and industrial compiler development. Within the realm of the LLVM compiler framework, LLVM IR plays a crucial role as it serves as an essential language that facilitates program analysis and transformation. Positioned strategically between source code and machine code, LLVM IR acts as a universal conduit, allowing various programming languages to leverage the optimization and code generation capabilities inherent in LLVM. What makes LLVM-IR truly remarkable is its ability to reside close to machine code while maintaining its high-level expressiveness and remaining platform-independent. The flexibility of LLVM shines through its compatibility with numerous frontends (including C, C++, Rust, and Swift) and backends (targeting architectures such as x86, ARM, and RISC-V), which are seamlessly unified through their shared use of the dependable LLVM-IR [4].

The LLVM-IR strictly adheres to the principles of a typed, Static Single Assignment paradigm. This adherence guarantees that each variable has a unique assignment and remains immutable after being assigned. The adoption of SSA form in LLVM-IR enhances code comprehension and enables significant optimization benefits. Additionally, the use of SSA form simplifies data flow representation, thus facilitating advanced compiler optimizations. As a result, LLVM-IR emerges as a powerful intermediate language for program analysis and transformation purposes [37].

The LLVM IR incorporates a deliberate typing system that guarantees precise data type definitions for each variable, providing enhanced compiler safety and optimization potential. Within the LLVM IR framework, there is extensive support for a wide range of fundamental data types including integers (ranging from i1 to i64), floating-point numbers (such as float and double), pointers, arrays, structures, and function types. This comprehensive set of data types allows for effective static analysis, optimization techniques, and code generation in the LLVM toolchain.

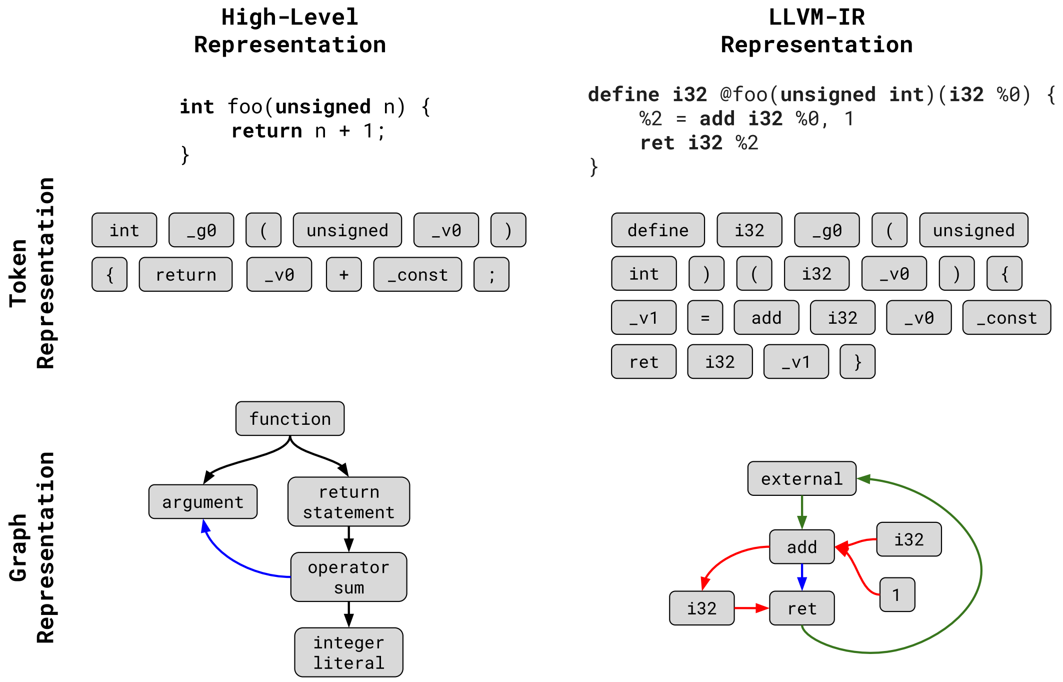
The LLVM module is a fundamental component of the LLVM-IR, serving as a comprehensive encapsulation unit for code, data, and metadata within a compilation unit. It organizes all relevant components related to a single compilation unit into a self-contained entity for code representation. Resembling assembly language in structure, LLVM-IR is organized into functions, basic blocks, and instructions. Functions are crucial units of code with their own set of basic blocks. These cohesive segments of code provide a linear flow of execution without branching instructions. In the LLVM-IR, instructions serve as fundamental components of functionality. Each instruction is defined by an opcode and input data. Notably, LLVM-IR opcodes are platform-independent, distinguishing them from conventional assembly opcodes.

This platform independence guarantees that LLVM-IR remains impartial to particular computer architectures or Instruction Set Architectures. Moreover, LLVM-IR introduces intrinsics, which are specialized functions that play a crucial role in representing custom behaviors and high-level code patterns. These intrinsics allow for the expression of low-level operations, enabling advanced optimizations by providing insights into specialized hardware capabilities or runtime behaviors. The LLVM Intermediate Representation encompasses a specific class of functions known as intrinsic functions or LLVM-IR intrinsics. These intrinsics embody low-level operations that are typically not expressible in the regular LLVM-IR language and act as a bridge between high-level language constructs and low-level hardware operations. By exposing functionality provided by target hardware or runtime environments, these intrinsics offer hints to the compiler for optimization purposes. In brief, the incorporation of LLVM-IR intrinsics in the LLVM framework enables efficient generation of machine code for different target architectures and hardware platforms. The modular organization and platform-agnostic opcodes of LLVM-IR, along with its versatility in supporting a wide range of programming languages and compiler optimizations through the use of intrinsics, make it an essential component in program analysis, transformation, and synthesis within compiler technologies like LLVM.

2.1.2 Source code structure analysis

The representation of source code is an essential aspect in numerous machine learning and compiler optimization tasks. There are two primary approaches to representing source code, regardless of whether it is written in its original high-level language [19] or translated into an intermediate representation such as LLVM-IR [4]. These approaches include token-based representations, which involve sequencing tokens within the input data, and graph-based representations that encompass a graphical representation of the code structure.

An essential approach in analyzing source code is token-based representation, which portrays the code as text [3]. This method treats source code as a means of human communication using tokens [2]. The process consists of two main steps: source rewriting and tokenization. To ensure accurate modeling of source code, the process of rewriting involves normalizing the input source by parsing it and removing elements that do not contribute to this task. Such elements may include comments, conditional compilation statements, or artifacts unrelated to accurately representing the source code (e.g., variable and function names). The purpose of source normalization is to simplify the model's learning process by eliminating trivial



```
int foo(unsigned n) {
    return n + 1;
}
```

```
define i32 @foo(unsigned int)(i32 %0) {
    %2 = add i32 %0, 1
    ret i32 %2
}
```

Figure 2.2: Summary of representation techniques for a C function.

semantic differences in programs caused by factors like choice of variable names or presence of comments. The process of tokenization involves transforming the input code into a sequence of language syntactic elements, called tokens. These tokens are stored in a dictionary and each assigned a unique numerical identifier. Token-based representation of source code can be both beneficial and challenging from a machine learning perspective. Sequentially structured data has been extensively studied in machine learning, with powerful methods available that yield impressive results on such data. The ordered sequence of tokens also accurately reflects the original ordering as written by the programmer, which provides valuable insights into their thought process during coding analysis. However, this strict ordering may impose constraints on representing code fragments that could potentially execute differently without such constraints.

Decades of research in the field of compilers have revealed that certain data structures provide a more comprehensive view of code optimization compared to simple token sequences. Graph-based program representation aims to capture intrinsic properties specific to the source code being represented, based on an appropriate strategy for representing the input program [18]. In terms of high-level language representation, graph-based approaches tend to expose the grammatical structure of the input program to machine-learning models. One common approach is representing the input program as its Abstract Syntax Tree (AST) [8]. The AST represents the abstract syntactic structure of text written in a formal language and each node signifies a construct found within this text. Moreover, to enhance the AST's effectiveness in tracking data dependencies and manipulations on the same data, additional data-flow information can be incorporated. This enrichment involves creating a labeled digraph known as a dataflow-enriched AST graph. In this graph, nodes are labeled as Declarations, Statements, and Types similar to those found in the Clang AST. The edges of this graph have two types of labels: type AST (representing child-of relationships within the AST) and dataflow (representing use-use relationships for variables). Moreover, sources can be translated into graphs to represent their intermediate representation. This graph consists of assembly-like statements commonly found in LLVM-IR. The edges of the graph connect the vertices based on different relationships that exist between pairs of edges. For instance, ProGraML [18] offers a comprehensive example of representing sources at the LLVM-IR level using a graph representation. In this approach, a full-flow graph is constructed by adding a vertex for each instruction and connecting them with control flow edges. To maintain order, all control edges are labeled with ascending numeric positions according to their sequence in the list of outgoing control edges from each vertex. Additionally, constant values and variables introduce extra vertices in the graph structure. To capture the relationship between constants and variables with their corresponding instructions, data-flow edges are introduced. These edges link variables and constants to the instructions that utilize them as operands, as well as instructions to the produced variables. Each unique variable and constant is represented as a vertex in ProGraML [18] graph-based representation, implicitly preserving the scope of variables. In contrast to tokenized machine learning approaches, this ensures that variables from different scopes are always mapped to distinct vertices and can be distinguished accordingly. Additionally, such a representation enables distinction between constant values with similar textual representations in source code. Similar to control edges, data edges include a position label that encodes the operand order for each instruction. Furthermore, the connection between a statement that invokes a function and the invoked function is represented by call edges. A call edge is created from the invoking statement to the entry statement of the function. Return call edges are included from all terminal statements within a function to the invoking statement. Unlike control and data flow edges, position labels are not used for call edges since there is no specific ordering required among different calls to a given function. Figure 2.2 shows the different representations possible for a small C function, both as a sequence of tokens and as a graph, depending on the abstraction level at which the function is analyzed.

2.1.3 Source Code Representation

The process of source code representation involves transforming the source code into a numerical form that is appropriate for input into a neural network model designed to solve the downstream task. There are two main approaches to representing source code: manual feature extraction and automatic representation learning through machine learning techniques.

Manual features extraction

Manual feature extraction in the field of machine-learning-based source code analysis relies on the expertise of compiler writers or domain experts who manually identify and select features that are relevant to solving a given problem [31]. One notable work in this area is [30], which presents predictive modeling techniques for determining whether it is beneficial to run OpenCL [52] code on a GPU or OpenMP code on a multi-core host. The predictor utilizes selected code features chosen by the compiler writer, summarizing costs associated with mapping decisions. During compilation, the paper analyzes the OpenCL code and extracts information about operations' numbers and types. Moreover, in addition to assigning higher importance to double-precision floating-point operations compared to single-precision ones, the memory access patterns are also analyzed to determine if accesses to global memory are coalesced or not. The number of control flow operations in an application is potentially relevant to functions mapped on GPUs, although it was not considered in [30]. Lastly, it is important to note that synthesizing raw features into aggregated ones can provide more informative insights compared to analyzing them individually. This approach involves combining and normalizing multiple features, allowing for a deeper understanding of the data.

Deep Learning-based features extraction – End-to-end learning

Historically, feature extraction in the field of machine learning has primarily relied on extracting significant aspects of data according to human expertise. However, when there is a lack of expert guidance, manually crafted features may not be as relevant. This can result in suboptimal performance for artificial neural networks [31]. To address this issue, recent advancements have steered away from relying on domain experts to manually extract features. Instead, the emphasis is now placed on empowering machine learning models to autonomously handle feature extraction tasks. This transition involves two main approaches. In one approach, known as end-to-end feature learning, the model starts with a random representation and refines it during training to address the specific task [3]. The other approach involves pre-training the model using representations that serve as a foundation for improving its ability to solve the intended task more effectively and accurately [5][55].

In the field of end-to-end representation learning, there is a technique where each unit of data is initially assigned a random embedding. This embedding is then refined by the network as it solves the downstream task during training. Notably, two research works by Cummins et al. [19], and Barchi et al. [4] demonstrate how this strategy can be applied to machine-learning models for source code analysis. These papers demonstrate similar techniques in which token representations are learned directly through an embedded layer during the end-to-end training process of machine learning models. Both methodologies represent code as a sequence of tokens and analyze source code at different abstraction levels, with [19] focusing on high-level OpenCL code and [4] analyzing LLVM-IR. In both cases, tokens in the vocabulary are mapped to unique integer values during encoding. To mitigate the limitation of sparse data representation caused by arbitrary integer values, an embedding layer is used. This layer translates tokens in a sparse, integer-encoded vocabulary into a lower-dimensional vector space. By doing so, semantically related tokens such as *float* and *int* can be mapped to nearby points within this space. The Embedding Layer acts as the first layer of the network, receiving sequences of token indexes and projecting each element into the embedding space. Consequently, it produces a list of sequences that consist of vectors belonging to the embedding space. Each token in the integer encoded vocabulary is mapped to a vector with real values through this embedding layer [3]. Given a vocabulary size V and embedding dimensionality D, during training, an embedding matrix W_E is learned with dimensions $\mathbb{R}^{V \times D}$. The token-indexes projection is determined by the weights of the Embedding Layer. Initially, these weights are randomly initialized, resulting in a random starting condition for the projection in the embedding space. As training progresses, these weights are updated. The output sequence from this embedded space, known as token-points, can then be passed on to subsequent layers of the neural network.

In recent advancements in machine learning-based source code representation, there has been an increasing interest in an alternative approach that involves pretraining a source code representation. This approach aims to create a representation that is independent of the specific downstream tasks. By doing so, it allows for a clear separation between the network responsible for assigning representations to data elements like programs, functions, or instructions, and the network dedicated to solving particular downstream tasks. To illustrate this approach further, we will discuss two state-of-the-art methods for constructing LLVM-IR statement embeddings: *inst2vec* [5] and *IR2Vec* [55].

Deep Learning-based features extraction – Code Embedding

The methodology known as *inst2vec* [5] presents a robust and effective technique for generating embeddings of LLVM-IR instructions. It is built on the theoretical framework of Neural Code Comprehension (NCC), which aims to represent code semantics in a manner that can be easily learned. NCC employs a versatile processing pipeline that transforms code from different source languages into statements written in the LLVM Intermediate Representation format. These statements are further transformed into contextual Flow Graphs, which capture both data-flow and control-flow aspects of the code, including loops and function calls. By utilizing this structure, *inst2vec* [5] trains an embedding space specifically designed for individual statements, allowing them to effectively address various high-level tasks in machine learning models. The NCC approach assumes that statements appearing in similar contexts have comparable meanings. In the *inst2vec* method, LLVM intermediate representation instructions are considered as statements and the context is defined using a contextual flow graph (XFG). The XFG represents both data-flow and control-flow relationships between pairs of LLVM-IR instructions through a directed multi-graph. Nodes in the XFG can be variables or label identifiers like basic blocks or function names, while edges represent either data dependence carrying an LLVM-IR statement or execution dependence. Based on this concept, *inst2vec* utilizes the skip-gram model to train an embedding space for individual statements. This technique is commonly employed in natural language processing to train word embeddings such as *word2vec* [41].

IR2Vec [55] is a robust and scalable encoding infrastructure that represents programs as distributed embeddings in continuous space. While *inst2vec* [5] utilizes the NCC framework to learn LLVM-IR instruction embeddings, *IR2Vec* [55] introduces an innovative approach for learning embeddings of LLVM-IR seeds. These seeds serve as the fundamental units of information within an LLVM-IR instruction. By composing seed embeddings hierarchichy, *IR2Vec* can generate embedding vectors not only for instructions but also for basic blocks, functions, and ultimately entire programs. This distributed embedding is achieved through a combination of various representation learning techniques and comprehensive flow information analysis. These approaches enable *IR2Vec* to effectively capture both the syntax and semantics of input programs in a rigorous academic manner while maintaining scalability. In addition, *IR2Vec* harnesses the capabilities of the LLVM-IR language format to generate embeddings that can be applied across multiple programming languages and machine architectures. This versatility makes them highly suitable for academic research purposes. To effectively represent entities within the intermediate representation, relationships are established and their representations are learned through a seed-embedding vocabulary. The core training procedure for seed embeddings lies at the heart of *IR2Vec* [55]. Each LLVM-IR instruction is deconstructed into its fundamental components: opcode, type, and arguments. While opcodes and types belong to finite sets, arguments are classified into five distinct categories: variables, pointers, and constants. In order to categorize arguments, *IR2Vec* proposes a classification system that includes three key relationships for describing an LLVM-IR instruction. The *TypeOf* relationship associates each opcode with its corresponding type, providing information on the nature of the operation. Meanwhile, the *NextInst* relationship captures control-flow details by linking together successive opcodes. Lastly, the Arg_i relation connects an opcode with its i-th argument class. To facilitate academic expansion, *IR2Vec* [55] extracts code triples from each LLVM-IR instruction: these triples consist of the instruction's opcode and type connected via a *TypeOf* relation; control-flow successors linked through *NextInstr*; and multiple Arg_i relations connecting an opcode to each of its argument classes. To train seed embeddings in *IR2Vec,* source files from the input dataset are first converted into LLVM-IR format before generating code triples for each instruction. The *TransE* [7] model is utilized to acquire the initial embedding, which generates knowledge-base embeddings by interpreting relationships as translations on entity embeddings in a lower-dimensional space. In this particular context, relationships can be depicted as translations within the embedding space: if there is a relationship between entities, then their respective embeddings should exhibit proximity to one another with the addition of a vector that corresponds to the specific relationship involved. The outcome of this learning process forms a dictionary containing embeddings of known entities referred to as Seed Embedding Vocabulary. This provided dictionary serves as a foundation for generating multiple levels of program-specific embeddings. At the highest level, symbolic encodings are constructed through coarse-grained instruction representations derived from instances found within the Seed Embedding Vocabulary set. Additionally, *IR2Vec* presents an approach for augmenting LLVM-IR instruction Symbolic encoding by incorporating *use-def* relations and reaching definitions to capture flow-aware characteristics inherent in instruction encoding patterns. To aggregate alive instruction embeddings and compute the embedding of basic blocks, liveness analysis is carried out on basic blocks. This aggregation process proceeds by combining basic block embeddings to generate LLVM-IR function embeddings.

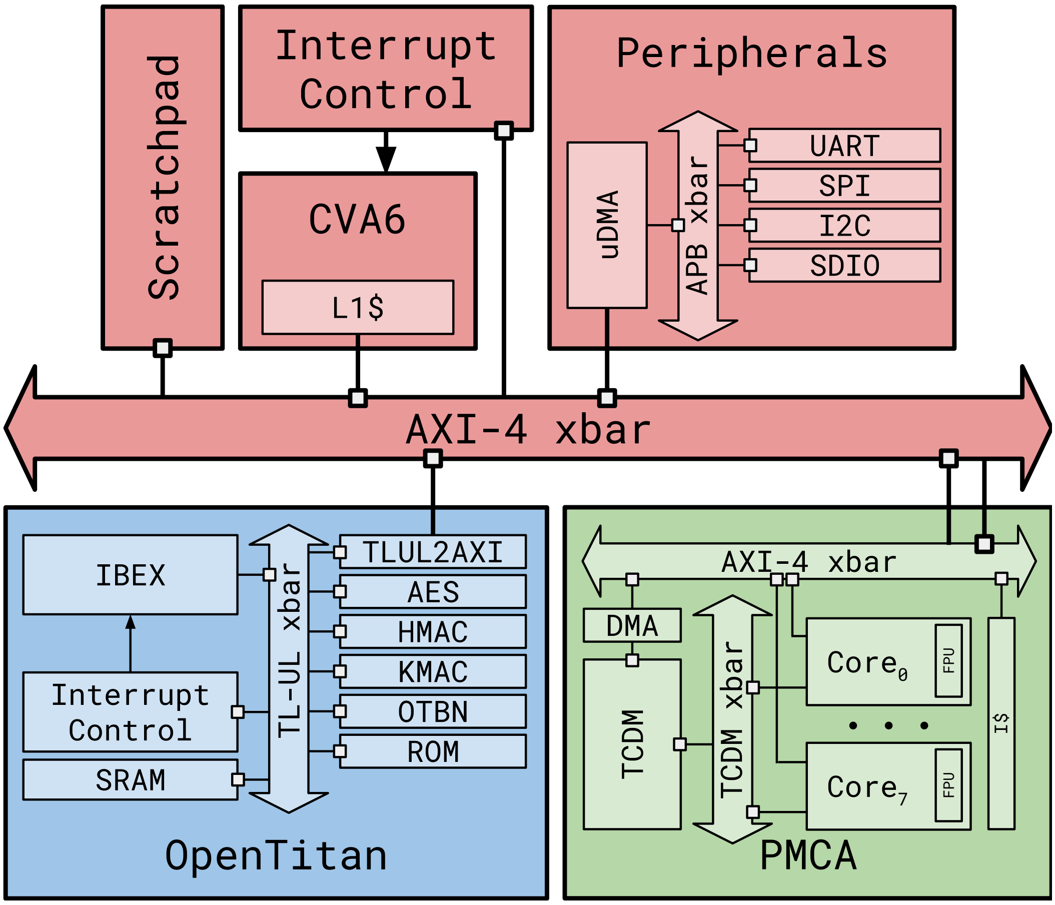


Figure 2.3: Architecture of the reference RISC-V system-on-chip used.

2.2 Architecture of the reference RISC-V platform

Most of the contributions of this thesis work have used a testbed built around PULP, a modular and heterogeneous RISC-V platform composed of a Linux-capable host-domain, described in Section 2.2.1, a general-purpose parallel cluster of 32-bit RISC-V core, described in Section 2.2.3, and a Root-of-Trust featuring a set of cryptographic accelerators and access to a private memory, described in Section 2.2.2. The architecture of the reference RISC-V platform is sketched in Figure 2.3.

2.2.1 CVA6 and Host Domain architecture

The host domain includes the CVA6 core, a scratchpad memory, and the peripheral domain. CVA6 [57] is a 64-bit RISC-V core that fully implements the Integer (I), Multiplication/Division (M), Atomic (A) and Compressed (C) extensions, which implement integer arithmetic, atomic memory transactions, and compressed opcodes. It has optional support for floating point extensions F and D. The core features three privilege levels - Machine (M), Supervisor (S), and User (U) - to ensure compatibility with Unix-like operating systems. The CVA6 core implements a six-stage pipeline consisting of PC Generation, Instruction Fetch, Instruction Decode, Issue Stage, Execute Stage, and Commit Stage. The first two stages serve as the frontend by selecting the next program counter for execution and handling fetch logic along with branch prediction. The branch prediction mechanism is vital for reducing the negative effects of control flow stalls on instruction per cycle. It employs three methods to predict the next program counter: a branch history table (BHT), a branch target buffer (BTB), and a return address stack (RAS). The Decode stage takes care of realigning and decoding instructions before storing them in an issue queue. In the Issue Stage, instructions are issued to the execute stage once all operands are ready using components such as the issue queue, scoreboard, and reorder buffer. The Execute Stage houses functional units including fixed latency units and variable latency units. One notable functional unit is the Load-Store Unit which hosts the memory-management unit (MMU) and interfaces with a private L1 data cache. The MMU of CVA6 offers hardware address translation to support the functioning of an operating system. The processor features separate and customizable data and instruction translation look-aside buffers, which are examined for a valid address translation with every access to instructions or data. Both data and instruction caches are virtually indexed and physically tagged and their set-associativity and cache-line size can be adapted. At synthesis time it is possible to choose between the implementation of a Write-Back or Write-Through cache, depending on the need of the target system to be implemented. If no valid translation is found, CVA6's hardware page table walker interrogates the main memory to acquire a valid address translation. Finally, in the Commit Stage instructions are read from ROB for committing purposes while also updating register files and resolving write-back conflicts through ROB mechanisms. The scratchpad memory is meant to store data to be shared with off-chip peripherals, or the support efficient communication between the CVA6 host core and the programmable cluster. The peripheral domain, implements many common embedded I/O interfaces and controllers, such as HyperBUS, I2C, (Q)SPI, CPI, SDIO, UART, CAN, PWM, and I2S. Additionally, it also features the uDMA, a controller intended to autonomously support data transfers between I/O peripherals and the scratchpad memory. Finally a 64-bit AXI crossbar [36] orchestrates all the communications in the SoC.

2.2.2 Root-of-Trust architecture

OpenTitan [16] is an open-source silicon RoT that implements a secure enclave aiming at securing sensitive data against hardware attacks, tampering, counterfeiting, and enabling the implementation of security mechanisms such as secure boot and remote attestation. The hardware architecture of OpenTitan includes (i) a secure microcontroller, (ii) on-board private memory, and (iii) a set of cryptographic accelerators.

The secure microcontroller is Ibex, an open-source 32-bit RISC-V core optimized for low-gate count, and designed for embedded control applications [21]. Ibex supports the Integer (I) or Embedded (E), Integer Multiplication and Division (M), Compressed (C), and B (Bit Manipulation) extensions. The core utilizes a two-stage pipeline architecture, consisting of an Instruction Fetch stage and an Instruction Execution stage. During the Instruction Fetch stage, instructions are retrieved from memory through a prefetch buffer that can fetch one instruction per cycle, if the instruction side memory is ready. The fetched instruction is then decoded and immediately executed in this same stage, with register read and write operations taking place as well. If there are any multi-cycle instructions, they will cause stalls in this stage until they are completed. Additionally, there is an optional Write-Back stage that can be enabled if needed to enhance core performance by reducing stalls on memory accesses at the cost of a slight increase in pipeline control logic complexity. Ibex is integrated into a wrapper that converts its interface to the TileLink Uncached Lightweight (TL-UL) interconnect employed by the platform.

Cryptographic hardware accelerators enhance system performance by efficiently executing compute-intensive security primitives. They prevent cryptographic computations from becoming a bottleneck and support key generation, buffer encryption/decryption, signature generation/verification, and hash calculation operations commonly used in the field of security. These accelerators include HMAC and KMAC for digest and hash calculations, Advanced Standard Encryption for symmetric cryptography operations, and OpenTitan Big Number Accelerator (OTBN) designed for RSA and Elliptic Curve Cryptography algorithms.

The OpenTitan system includes a 128KB SRAM memory and an embedded flash memory that has been enhanced with Error Correcting Code and data and address scrambling to improve security and reliability. In the reference system, OpenTitan is integrated into the SoC, allowing it to access memory through a custom TileLink-to-AXI bridge [15]. Communication between the host domain and RoT is facilitated by a shared-memory mailbox. This mailbox consists of general-purpose memory-mapped registers designed for data sharing. Additionally, it also features two specific registers - Doorbell and Completion - that are used to send interrupts to both the Ibex security microcontroller and CVA6 host core. OpenTitan also includes a peripheral subsystem for communication with the host domain and external hardware modules. This subsystem supports UART, USB, I2C, SPI, and GPIO interfaces.

2.2.3 Programmable Multi-Core Accelerator architecture

The Programmable Multi-Core Accelerator (PMCA) [47] is a collection of 8 RISC-V cores based on the CV32E4 microcontroller [28]. The PCMA is used by the host to execute kernels that require intensive computations. Each core in the cluster follows an in-order, 4-stage pipeline and supports the RISC-V instruction set architectures I, M, and C. Additionally, they incorporate extensions specifically designed for DSP and machine learning workloads to fulfill computational requirements [27]. The core is enhanced with a custom RISC-V extension that allows for efficient manipulation of both real and complex numbers. This greatly improves the core's ability to perform DSP-centric operations more effectively. Furthermore, the fetch stage of the core now supports zero-overhead hardware loops, making it possible to execute repetitive tasks commonly found in DSP and machine learning workloads without any additional overhead typically associated with traditional loop constructs. In addition, efficient array manipulation is facilitated through post-modified load and store instructions which enable single-cycle multiply and accumulate operations as well as single-cycle complex multiplication. Moreover, dedicated instructions are provided for efficient rounding, normalization, and clipping. To enhance instruction-level parallelism, the core's instruction set is expanded to include SIMD instructions. These instructions reduce the width of operands, allowing for double or quadruple the number of operations per cycle. For integer numbers, precision can be reduced to 8-bit, and for floating-point numbers, it can be reduced to 16-bit. The core's instruction set is enriched with bit-manipulation-oriented instructions, including bit insertion and extraction operations. These instructions improve code compactness and execution speed for bit-level manipulations in control-oriented code. Floating-point computation is handled by 8 FPUs supporting FP32 and FP16 with SIMD support [42].

The cluster's memory architecture incorporates a two-level instruction cache system, consisting of 512 bytes dedicated I-caches for each core and a 4 KB shared I-cache, optimizing instruction access. To avoid data cache overhead, the cluster employs a shared 128 KB scratchpad memory accessible by all cores. Access to the L1 memory is supported by a highly optimized interconnect which ensures low-latency memory access and overall performance efficiency.

The PMCA interacts with the host's AXI interconnect using two 64-bit AXI ports, one for master and one for slave communication. The cluster includes a DMA with an AXI port and four ports connected to the scratchpad, enabling high-bandwidth and low-latency transactions between the cluster memory. An event unit is dedicated to efficient event management and parallel thread dispatching on a fine-grained level [29].

2.3 Security and Binary Exploitation

In today's society, Internet of Things devices have become an integral part of our daily lives. They are used in various critical areas such as autonomous vehicles, power grids, and healthcare. It is crucial to prioritize strong security measures and address cybersecurity risks due to the growing presence of these devices. Events like security breaches in vehicular systems and the well-known Mirai botnet, which involved a widespread distributed denial-of-service attack solely using IoT devices, highlight the weaknesses present in IoT systems. One of the challenges in securing modern IoT node firmware is that it is usually developed using programming languages such as C and C++ [11]. These languages give developers a lot of control over resource usage, but their manual memory management and adherence to typing rules also introduce security vulnerabilities. As a result, protecting embedded systems from various security threats becomes crucial. Of particular concern are software vulnerabilities that can lead to memory corruption [9], enabling unauthorized access, hijacking target processes, and causing significant disruptions.

2.3.1 Code-Reuse Attacks and Return-Oriented Programming

Control-flow hijacking attacks pose a significant threat to computer systems by exploiting memory corruption vulnerabilities. These exploits involve corrupting specific memory regions, such as the stack's return address, which enables attackers to redirect program execution towards malicious code called a payload. The objectives of these attacks include arbitrary code execution, privilege escalation, and unauthorized access to sensitive information. Historically, the technique of "stack smashing" [43] has been employed to carry out execution hijacking. This method exploits buffer overflows to manipulate the memory region of the stack by injecting malicious code called shellcode, which executes malicious actions as intended by the attacker. Additionally, it involves modifying and redirecting the return address stored in the stack to point toward the initial binary instruction of this injected shellcode. As a result, when the vulnerable function returns, instead of following normal behavior, program execution is redirected towards this injected code giving unauthorized control over program behavior. In order to counteract these threats, modern computer systems have implemented mitigation strategies. One such strategy is the Write xor eXecute principle, which ensures that a writable memory region, including the stack, cannot be marked as executable at the same time. However, despite extensive research on exploit mitigations, only a few of them are actually being used [54]. While defenses like Address Space Layout Randomization (ASLR), stack canaries [17], and Data Execution Prevention [51] combination protect the flow against code-injection attacks but they fail to fully prevent code-reuse attacks. Code-reuse attacks (CRAs) involve manipulating existing fragments of code for malicious purposes without introducing new code and have become increasingly prevalent in recent times. Common manifestations of code-reuse attacks are Return-Oriented Programming (ROP) [46], or Jump-Oriented Programming (JOP) [6], wherein attackers craft sequences of legitimate code instructions to execute their desired actions.

Return-oriented programming bypasses memory protection mechanisms like $W \oplus X$. Unlike traditional code injection methods, ROP attacks do not insert new code into the system. Instead, they manipulate existing snippets of code called gadgets that end with a return instruction and are present in the program's address space. By linking these gadgets together through control of the stack, an attacker can execute malicious behavior without introducing new code. This approach makes ROP attacks less detectable compared to traditional shell code injection techniques as they utilize pre-existing instructions as building blocks for their exploits. ROP is a technique that utilizes the stack pointer as a program counter, determining the sequence of instructions to be executed. Unlike traditional programming approaches where instructions are stored in the text segment, ROP constructs its program using the stack segment. Each ROP instruction on the stack corresponds to an instruction sequence located within the exploited program's memory. The progression of execution is dictated by the stack pointer, which determines which ROP instruction will be executed next. When a return instruction is encountered, it updates the core's program counter with the address of the subsequent gadget address for sequential execution of attacker-crafted gadget chains to proceed smoothly. ROP is a technique that involves arranging gadgets to carry out an attacker's intended actions. These gadgets are organized within the program's memory, and the stack pointer is redirected to execute the first gadget. While buffer overflows are commonly employed for this purpose, other methods such as overwriting function pointers can also be utilized.

2.3.2 Control-Flow Integrity

Control-Flow Integrity (CFI) [1] has emerged in the state of the art literature on security as a technique to constrain the possible control-flow transfers of an application to those that have been specified by the developer and alert the platform runtime if any control-flow violations occur. While the ISA of a machine allows indirect control-flow transfers to target any executable address in memory [26], in the high-level languages the valid targets are restricted by language constructs. In practice the target of any jump will always be the first executable instruction of a basic block of code, such as functions or methods. CFI prevents code-reuse attacks because they would cause the program to transfer control to a target which is illegal under CFI. In practice, CFI closes the gap between ISA capabilites and high-level language semantics by constraining control-flow transfers target locations.

Most CFI mechanisms follow a two-phase process: analysis and enforcement [59]. During the analysis phase, the control-flow graph (CFG) of the target program is computed by a static binary analyzer. Since the limitations of static program analysis may lead to an overapproximation of the control-flow transfers that can actually take place at runtime, limiting the security of the enforced CFI policy, some methodologies enrich the analysis phase by running the binary to be protected with real-world data, extract the execution traces using available tracing support, such as Intel IPT, and use such information to prune non-essential edges from the static CFG [58] [13]. The enforcement phase ensures that control-flow transfers whose targets are computed at runtime, such as indirect branches and return instructions—correspond to edges in the CFG produced by the analysis phase. These targets are commonly separated into two categories: forward, which give control to a new location inside a program, and backward, which return control to a previous location. ISAs usually offers two forward control-flow transfer instructions: call and jump. While a jump simply updates the program counter, moving execution to another target address, a call instruction pushes the address of the immediately following instruction onto the stack, so that whenever a return instruction is executed, such address is fetched back into the program counter of the machine and execution can resume from the next address before the call instruction. Forward control flow can be direct or indirect, depending on whether the target address can be statically determined at compile time (direct) or is computed at runtime (indirect). A return instruction is the symmetric counterpart of a call instruction, and a compiler emits function prologues and epilogues to form such pairs. Returns are always, and intrinsically, indirect control flow changes. CFI enforcement schemes should also take into account global exception-triggered control-flow manipulation, that is, interprocedural control flows that require unwinding stack frames on the current stack until a matching exception handler is found.

CFI can be enforced both with software and hardware approaches. Assuming that code is static and immutable, software solutions instrument indirect control-flow transfers at compile time through a modified compiler, or during execution through dynamic binary translation [45]. The types of indirect transfers that are subject to such validation and the number of valid targets per branch varies greatly between different CFI defenses. Hardware approaches extend the SoC architecture to provide runtime-checking capabilities for the executed software. These solutions can be categorized into three main approaches: (i) ISA extensions, (ii) hardware monitors, and (iii) programmable coprocessors. ISA extension techniques [22] introduce custom opcodes for conducting security checks during program execution. While these techniques typically have minimal runtime overhead, they require ad-hoc binary toolchains and invasive modifications to the processor pipeline to accommodate these new instructions. Additionally, all software must be rebuilt, and legacy binaries cannot benefit from this protection. Hardware monitors [50] utilize custom-designed IPs closely integrated with the processor pipeline to monitor control-flow instructions and implement security features like shadow stacks or jump tables directly in silicon. These techniques are transparent to the executed software and do not require extensive modifications to the core architecture. However, they lack flexibility in dynamically updating the CFI enforcement policy, requiring the creation of a new monitoring system from scratch. Security coprocessors [23] [24] rely on a dedicated semi-programmable core that enables the implementation of the desired policy in software. These co-processors use a reserved side-channel to communicate control-flow information. This approach allows for the implementation of customized policies on the co-processor, which operates in parallel with the main core. However, a co-processor capable of implementing arbitrary policies is typically larger than a customized hardware IP, therefore, there is a significant increase in area overhead.

3 Machine Learning-based Device Configuration

3.1 Introduction

This chapter is dedicated to exploring the feasibility of using static source code analysis to determine the optimal configuration for an embedded RISC-V core cluster, with a focus on minimizing power consumption during compilation. This innovative approach involves extracting unique static features from the LLVM-IR [37] representation of a code segment, eliminating the need for dynamic profiling or runtime assessments. Additionally, it includes evaluating any differences in precision between system configurations derived from static source code analysis and those obtained through dynamic attribute extraction via profiling on the target platform.

Previous research in this field has recognized the significance of source code analysis for decisions regarding system configuration, such as parallelism mapping [56] and thread coarsening [39]. However, these studies have not adequately addressed the crucial concern of energy optimization specifically within ultra-low-power embedded architectures. Additionally, existing literature on power and energy modeling for parallel architectures has primarily focused on profiling-based features rather than predicting optimal scaling configurations (i.e., number of parallel cores) based solely on source code information. Therefore, the main objective of this study is to fill this gap by approaching it as a classification task where a classifier is trained to assign each computational kernel to its corresponding minimum-energy category.

In this chapter, I have made several technical contributions including: first, a dataset of kernels specifically designed for energy classification in parallel architectures. Each dataset sample has been ported to the PULP [47], a cluster of 8 efficient RISC-V cores, optimized for DSP applications, presented in Section 2.2.3. Secondly, I highlight the complexity of the energy classification problem and emphasiz that it cannot be simply treated as an extension of performance or speed-up classification. Lastly, through empirical experiments and analysis, I demonstrate the feasibility of source code-based energy classification techniques and quantified their accuracy compared to dynamic features-based methods.

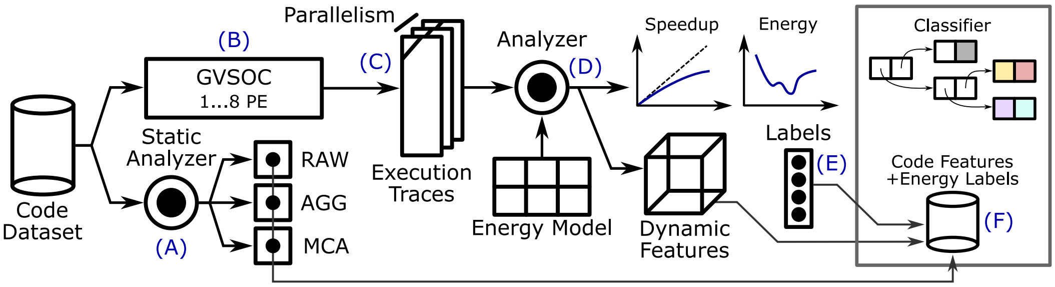


Figure 3.1: Workflow to identify the minimum energy parallelism on a PULP cluster and to define a dataset composed of static and dynamic features.

3.2 Methods

3.2.1 Methodology

In the field of embedded parallel processors, achieving software energy efficiency involves leveraging hardware parallelism [48]. As the application utilizes more cores, the runtime decreases, along with leakage energy, but dynamic power consumption increases. The purpose of this chapter is to demonstrate that by using a machine learning model trained on static source code information, it is possible to determine the optimal configuration for minimizing energy consumption on parallel embedded microcontrollers. Figure 3.1 details the proposed approach which consists of the following steps:

(A) A preliminary features extraction activity is conducted on all samples in the dataset through static source code analysis. Comprehensive information regarding the dataset construction and machine learning features can be found in Sections 3.2.2 and 3.2.4.

(B) Each sample in the dataset is analyzed using GVSOC [10], a cycle-accurate simulator of the PULP cluster. It provides detailed execution traces for tracking opcodes executed, memory transactions, active wait cycles, and cores idleness caused by clock gating.

(C) All samples are simulated eight times using an increasing number of the cores available in the PULP cluster. This allows an extraction a runtime trace of the activity of each core during each cycle of the duration of the benchmark run, providing valuable insights into system performance and efficiency.

(D) The execution traces are combined with the energy model presented in Table 3.1. This approach enables the assignment of an accurate energy cost to each sample

based on the number of utilized cores. For a detailed description of the energy model, please refer to Section 3.2.3.

(E) In order to minimize energy consumption, a detailed analysis is conducted on the eight traces obtained for each benchmark. This analysis allows labeling each sample with the number of execution cores that would yield optimal results in terms of energy efficiency.

(F) The collection of labelled samples, each with its specific set of static features, represents the dataset used for training the decision tree algorithm.

3.2.2 Dataset description

A carefully defined collection of parallel programs to be measured and analyzed has been established. To express kernel parallelism, I have chosen OpenMP [20], a widely-used programming model for shared-memory architectures that is supported by an increasing number of platforms including PULP. It should be noted that many embedded research-oriented architectures do not fully implement the OpenMP standard; instead, they provide a subset of functionalities specifically designed to support common scenarios. The PULP cluster runtime implements a partial version of the OpenMP standard which does not support tasking and only provides a limited selection of loop scheduling policies. Consequently, in this study, we had to meticulously customize the application kernels comprising the dataset and often exclude publicly available OpenMP datasets due to limitations in functionality provided by PULP's implementation. In terms of memory allocation, the PULP [47] runtime offers a comprehensive but non-standard range of interfaces for allocating on-cluster data. This allows for efficient access to cluster private memory without requiring explicit programming for DMA transfers from off-cluster memory, which is the default destination for dynamic memory allocation. Additionally, adjustments have been made to the benchmarks to make them adaptable based on the type of data used during computation. These modifications are necessary because embedded systems often have constrained hardware resources that can affect program behavior depending on the processed data variables being utilized. This becomes particularly relevant when handling floating-point operations that compete over resources such as FPUs across cores.

Table 3.1: The energy model used to label the dataset. The energy consumption of every system-on-chip component is modeled by its leakage and and the dynamic energy required per cycle, depending on its state.

Operating Region	Energy [fJ]
Processing Element	
Leakage	182
NOP	1212
ALU	2558
FP	2468
L1	3242
L2	1011
CG	20
FPU	
Leakage	191
Operative	299
Idle	0
Other Cluster Components	
Leakage	655
Active	2702

Operating Region	Energy [fJ]
Memory Bank L1	
Leakage	49
Read	2543
Write	2568
Idle	64
Memory Bank L2	
Leakage	105
Read	2942
Write	3480
Idle	13
ICache	
Leakage	774
Use	4492
Refill	5932
DMA	
Leakage	165
Transfer	1750
Idle	46

3.2.3 Energy model

The energy estimation of the RISC-V cores in the PULP cluster was conducted using an energy model, as described in Subsection 3.2.1. The activity traces from the GVSOC simulator were utilized to extract information for this purpose. Details about the energy model can be found in Table 3.1. In terms of both leakage and switching activity, each component of the PULP cluster contributes to its overall energy consumption. The power consumed by processing elements is dependent on opcode classes executed as well as on active wait cycles performed. Furthermore, in order to minimize power usage during periods of inactivity, advanced low-power states such as clock-gating are employed. The models for memory, FPU, and DMA take into account the differences between active and idle power consumption. Additionally, the models also consider the energy costs associated with read and write operations. In addition to these considerations, further costs are taken into account for energy consumption related to circuitry not explicitly modeled within the PULP cluster. This includes factors like the cores-to-TCDM bus and the event unit responsible for managing power gating and interrupt dispatching.

3.2.4 Feature selection

This section provides a detailed description of the features that were taken into consideration for training the classification model. For this study, I utilize two ensembles of static source code features. The first ensemble consists of the raw and aggregate forms of the features introduced by [30]. Additionally, I incorporate a second set of features provided by LLVM-MCA, which is a machine code analyzer tool included in the LLVM compiler suite.

The authors in [30] explore six RAW metrics for static analysis of OpenCL [52] kernels, which were then condensed into four features used for the decision tree. However, when dealing with deeply embedded systems like PULP, not all the RAW metrics from [30] are applicable. Specifically, there is no difference between global and local memory accesses as we assume that all data resides in TCDM. This assumption is reasonable since architectures like PULP achieve maximum efficiency when data access occurs within on-cluster TCDM. The challenge in programming embedded devices similar to PULP lies in carefully managing DMA transfers from off-cluster memory to overlap transfers with computation efficiently. Furthermore, it should be noted that coalescing is not relevant in our case because scratchpad memories are not affected by access patterns. Additionally, the average number of work-items per kernel is a specific metric for the OpenCL programming model

Table 3.2: Description of the different class of static features used to predict platform configuration.

	Features		Description
	[30]	This work	
Raw	`comp`	`op`	# of ALU, FP and JUMP opcodes
	`mem`	-	Not used
	`localmem`	`tcdm`	# of accesses in on-cluster TCDM memory
	`coalesced`	-	Not meaningful on the PULP architecture
	`transfer`	`transfer`	Amount of data the kernel works on
	`avgws`	`avgws`	Average # of iterations in parallel regions
Aggr.	`F1`	`F1`	`transfer` / (`op` + `tcdm`)
	`F2`	-	Not used, depends on `coalesced`
	`F3`	`F3`	`avgws`
	`F4`	`F4`	`op` / `tcdm`
LLVM-MCA	-	`uOPSpc`	Micro operations issued per cycle
	-	`RBP`	Instructions per cycle
	-	`RPDiv`	Reverse block throughput
	-	`RPFPDiv`	Resource pressure on the divider port
	-	`RP0`	Resource pressure on the FP divider port
	-	`RP1`	Resource pressure on Port 0 (Misc)
	-	`RP2`	Resource pressure on Port 1 (Misc)
	-	`RP3`	Resource pressure on Port 2 (AGU, Load Data)
	-	`RP4`	Resource pressure on Port 3 (AGU, Load Data)
	-	`RP5`	Resource pressure on Port 4 (Store Data)
	-	`RP6`	Resource pressure on Port 5 (ALU, ALU Vec., LEA)
	-	`RP6`	Resource pressure on Port 6 (ALU, Branch)
	-	`RP7`	Resource pressure on Port 7 (AGU)

and does not apply to OpenMP codes. Instead, I suggest considering the average number of iterations that can be executed concurrently within the parallel regions of an OpenMP kernel. As for combining the RAW metrics into four AGGregate static features, I follow the framework described in [30] work and summarized in Table 3.2.

The LLVM framework [37] provides a static machine code analysis tool called LLVM-MCA. This tool measures the performance of machine code in a specific CPU, considering factors such as throughput and processor resource consumption. By modeling the execution engine of different microarchitectures, LLVM-MCA can provide insights into opcode dispatch to various execution units or "ports," assuming perfect branch predictions and cache hits. An important output of this analysis is the generation of metrics known as "port pressures." These metrics quantify how much

Table 3.3: Dynamic features used train the decision tree classifier.

	Features	Description
Dynamic	PE_idle	Fraction of cycles in which a core incurs in resource contention or in a multi-cycle instruction.
	PE_sleep	Fraction of cycles in which a core is in clock-gating.
	PE_alu	# of opcodes involving the usage of the ALU.
	PE_fp	# of opcodes involving the usage of the FPU.
	PE_l1	# of opcodes involving access to the TCDM.
	PE_l2	# of opcodes involving an access to off-cluster memory.
	L1_idle	# of cycles in which a TCDM bank is idle.
	L1_read	# of read request received by a TCDM bank.
	L1_write	# of write request received by a TCDM bank.
	L1_conflicts	# of contemporary requests received by a TCDM bank.

an instruction flow stimulates execution units. This study aims to explore whether these easily collectible features from the LLVM framework can be used as a static kernel fingerprint to improve the modeling capabilities of our decision tree classifier for our domain-specific problem solution.

3.3 Results

3.3.1 Test Bed

The GVSOC [10], a virtual platform included in the PULP-SDK, offers significant advantages for integration into development flows. Compared to RTL simulation, it provides fast and cycle-accurate performance. Additionally, the virtual platform generates execution traces that capture detailed information about the status of cluster components throughout program execution.

The power consumption numbers have been obtained through an analysis using Synopsys PrimeTime 2019.12 with a nominal voltage of 0.65 V and extracting value change dump traces from post-layout simulation in Mentor Modelsim 2008.06 for synthetic benchmarks. These numbers encompass both static and dynamic power components. By integrating these values, the energy consumption associated with a specific class of instructions can be determined since each benchmark focuses on a single instruction class.

We utilized GVSOC to capture the execution traces and determine the energy consumption during the execution of an OpenMP kernel on PULP. The traces consist of a record of events triggered by the various components simulated in the virtual platform. Each component is identified by its specific path within the architecture. To calculate energy, I employe trace analysis software composed of two modules: hierarchical listeners and a trace analyser. These listeners are grouped together within the `PULPListeners` class, which offers methods to access information about the platform and its components. `PULPListeners` comprises 8 `CoreListeners`, 16 `L1BankListeners`, and 32 `L2BankListeners` respectively registering with their corresponding paths on the trace analyzer for event capturing purposes. The GVSOC trace analyzer processes the GVSOC trace line by line, using regular expressions to extract information such as the event cycle number and the component that generated the event. This extracted data will be further analyzed by a listener. The CoreListeners receive events from `cluster/pe/insn` to analyze opcode traces and from `cluster/pe/trace` to identify clock gating regions and wait cycles. The `BankListeners` receive events from `cluster/l1/bank/trace` for analyzing write and read events on banks, as well as counting conflicts that occur when multiple requests are received in the same cycle. By analyzing the trace, it is feasible to exclude events that fall within a specific range of cycles. This process entails determining the cycle range in which the parallel code fragment (function `void kernel(...)`) resides. Within this designated region, I calculate the energy contributions for each component in the platform considering the features specified in Table 3.3 and using energy model mentioned in Section 3.2.3.

3.3.2 Dataset analysis

The dataset utilized in this study encompasses three different sets of benchmarks, comprising a total of 59 distinct C kernel programs. These benchmark suites were selected for the purpose of analysis and include *Polybench*, *UTDSP*, and *Custom*. Polybench is widely recognized as a comprehensive program set that evaluates polyhedral optimization functionality in compilers. *UTDSP* includes a series of kernels specifically designed to test optimization techniques targeting digital signal processors. In addition, an assortment of manually crafted kernels was incorporated into the dataset to simulate various memory access patterns, computational operations, and synchronization mechanisms.

Each individual kernel within the system is designed to handle specific types of data and process varying amounts of data. In the analysis, I primarily focus on 32-bit integers and single-precision floating-point numbers. The decision to exclude double precision floating-points stems from the lack of support for them in the processing elements within PULP [47]. Furthermore, I have chosen not to explore the effects of utilizing compact integer types like 16 or 8 bit integers until future research endeavors.

The execution of each kernel, instantiated with a specific type, is repeated multiple times with the different amount of processing data, for checking how problem size impacts energy efficiency. For each kernel, I test a problem size of 512, 2048, 8196, and 32768 bytes. The quantity and variety of chosen payload sizes have two advantages. On the one hand, it reflects a typical payload size suitable for the amount of computation in a parallel microcontroller of the power class of PULP. On the other hand, such a choice allows us to fit all the data the benchmarks work on in the scratchpad memory. In this way, I avoid the need to take into account DMA transfers from the off-cluster memory to the scratchpad, which would make the energy analysis notably more difficult. Under the assumptions above, the dataset of kernels we used to train and test the machine learning model is composed of 448 samples. The dataset shows a class unbalance between 5% and 15%, except for the class with label *8* which accounts for the 34.8% of the samples collection.

To assess the performance of the classifier, two key metrics were utilized: absolute accuracy and t-accuracy. Absolute accuracy is a commonly employed metric for evaluating machine learning classifiers. It measures the fraction of correct predictions made by the model in comparison to the total number of samples analyzed. However, the experimental setup also takes into consideration that in certain scenarios that may be acceptable from an engineering perspective, to select a number of processing elements that results in some energy wastage compared to the theoretical minimum. To address this concern, the concept of t-accuracy is introduced, which is similar to absolute accuracy but considers a prediction correct if the energy loss falls within $t\%$ of the theoretical minimum. In practical terms, a sample in the dataset is most efficient when parallelized with four processing elements, but the classifier predictions should be computed with six processing elements. This prediction is deemed correct if the additional energy wasted running that kernel with six cores instead of four is lower than $t\%$.

To ensure unbiased accuracy results, each training experiment in the following section employs a 10-fold stratified cross-validation method. Furthermore, for added reliability, every cross-validation is repeated 100 times using random seeds.

3.3.3 Optimal configuration selection

The objective of this research is to assess the potential of machine learning models in determining the optimal parallelism configuration for maximum energy efficiency

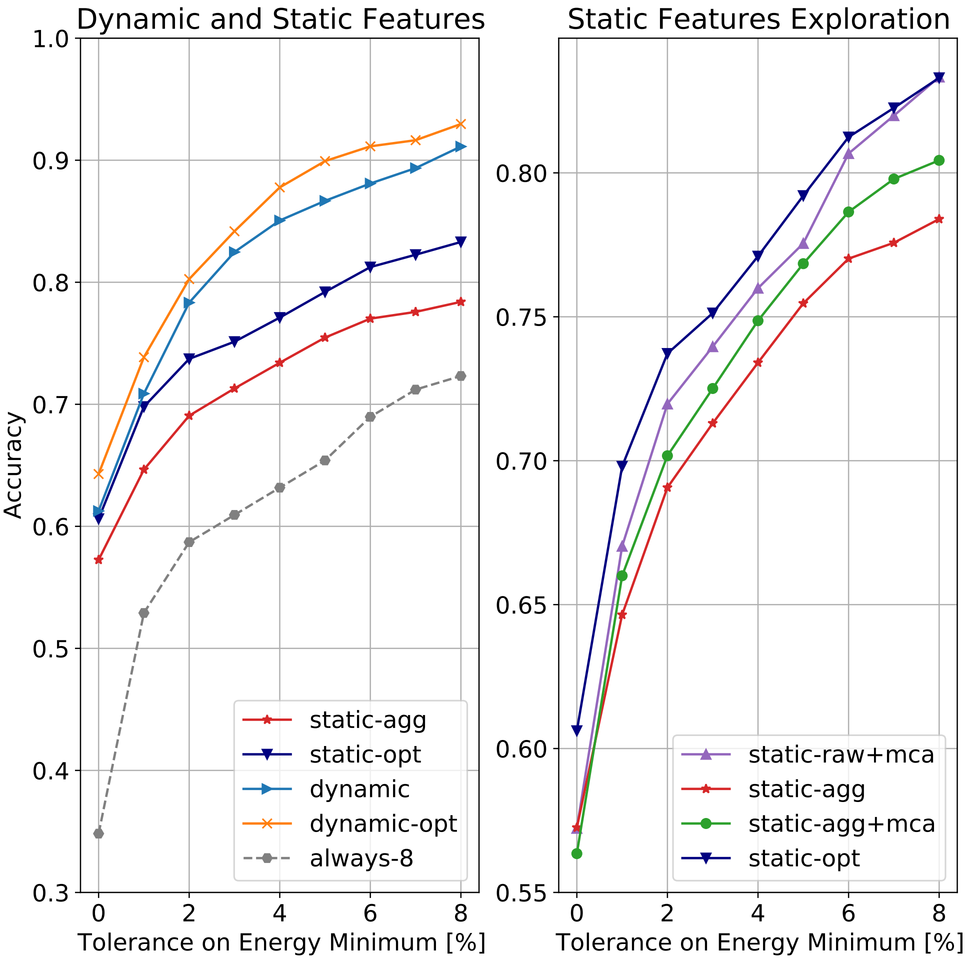


Figure 3.2: Classification accuracy obtained by the classification mode, as a function of the tolerance on the energy minimum. The left plot compares static and dynamic features with the naive "always-8" choice. The right plot shows the classification accuracy for different static features.

on PULP architecture. The study revolves around examining whether it is feasible to utilize solely static code analysis-based features as inputs for training a classifier. This investigation involves three key stages: i) Initial evaluation of aggregate static features ii) Analysis leveraging dynamic features obtained from GVSOC traces and identification of the most valuable classification features iii) Refinement of static feature optimization methods.

To begin with, the accuracy of the decision tree classifier needs to be assessed using the static features shown in Table 3.2. Initially, the aggregate set of features (F1, F3 and F4) is used, outlined in Table 3.2 and compare the findings with a naive classifier that utilizes all processing elements in the cluster (always-8). The comparison is shown in Figure 3.2 (left plot) where it is evident that the red line consistently outperforms the dashed grey line. In particular, when considering a tolerance of

Table 3.4: Static and dynamic features ranked by their importance score.

	Feature	PE	Importance
Static	F3	-	19.6 %
	F4	-	11.7 %
	F1	-	6.8 %
	RP4	-	3.5 %
	uOPSpc	-	3.2 %
	RP7	-	3.1 %
Dynamic	PE_sleep	8	19.6 %
	PE_sleep	2	11.7 %
	PE_idle	5	6.8 %
	L1_write	1	6.7 %
	L1_conflicts	6	4.1 %
	L1_read	8	4.0 %
	PE_sleep	5	3.5 %
	L1_conflicts	5	3.2 %
	PE_sleep	6	3.1 %
	PE_alu	6	2.3 %
	PE_sleep	7	2.1 %
	PE_idle	3	1.9 %

5% on wasted energy, the classification accuracy exceeds 75%. Exploring dynamic features plays a crucial role in identifying new necessary static features to enhance classification performance.

Then a similar experiments is conducted using dynamic features extracted from GVSOC [10]. As traces are utilized to calculate the energy consumption of a program, they provide the "ground truth" for determining the optimal energy parallelism. Considering that dynamic features are expected to be more informative than static ones, our goal is to identify the most optimal subset that yields improved classification results. The decision tree ranks dynamic features based on their importance. After conducting the analysis, a list of significant features, provided in Table 3.4, has been compiled.

One of the most significant features is the PE_sleep attribute, which accurately measures clock-gating cycles with a parallelism of both 8 and 2 cores. These values provide valuable insights into how the source code behaves under different levels of parallelism. Additionally, other notable attributes should also be taken into considerations such as PE_idle when utilizing five cores and L1_write operations without any form of parallelism. The former indicates wait cycles using half of the available parallelism, while the latter identifies memory writes that do not utilize any form of parallel execution.

The left plot of Figure 3.2 showcases the classification accuracy over 8 classes using different combinations of static features. The accuracy, without any energy tolerance threshold, is consistently around 57%. However, when allowing a 5% energy threshold tolerance, which is feasible in most cases, the classification accuracy increases to approximately 80%. By scoring and pruning less informative features used by the decision tree classifier, an "optimized" classifier can achieve an accuracy of 61% without a threshold and 79% with a 5% threshold over eight classes.

3.4 Conclusions

The problem of automatic source code configuration becomes increasingly relevant as architectures grow more complex and diverse. This study aims to predict the optimal number of cores for minimizing the execution energy of openmp kernels on deeply embedded architectures using static code analysis. Focus of this thesis is on the PULP architecture, an advanced parallel ultra-low-power embedded microcontroller. By incorporating dynamic features into the decision tree model, it is possible to identify the most promising features that enhance the robustness of the classifier based solely on static features. Ultimately, the results demonstrate that a decision tree trained with static source code features achieves a significant accuracy rate of 61%. Furthermore, when introducing a tolerance threshold of 5% for wasted energy during classifier evaluation, accuracy improves further and approaches 80%.

The first activity needed to further enhance this study is to expand the dataset coverage by including more kernels and exploring various parallel programming models. Additionally, any analysis should include DMA transfers and memory hierarchy to seek for a more complete and comprehensive platform representation. Furthermore, having a sufficiently large dataset, deep learning model can be exploited to avoid manual features engineering, augmenting the prediction capabilities of the existing solutions proposed in this research work.

4 Deep Learning-based Heterogeneous Device Mapping

4.1 Introduction

Heterogeneous computing platforms have gained significant importance in the modern era due to the changing landscape of semiconductor technology. The slowdown of Moore's law, which predicted a consistent increase in the performance of individual processors, paired with the end of Dennard scaling, which allowed processors to become more power-efficient as they grew in complexity, boosted the adoption of heterogeneous architectures in recent years, to meet the increasing computational demands of various applications [25]. Heterogeneous architectures, which integrate general-purpose processors and specialized hardware accelerators, are crucial for a wide range of applications such as artificial intelligence, scientific computing, mobile devices, and autonomous systems [34]. This architectural approach enables efficient utilization of hardware resources and optimization of performance and energy efficiency across various computing systems.

The emergence of state-of-the-art methodologies based on supervised deep learning models for source code analysis has provided new opportunities to enhance compilers to efficiently face novel challenges offered by heterogeneous computing [19, 18, 3, 8]. One of which is heterogeneous device mapping that refers to the problem of choosing the best computational unit, among the one available on a heterogenous platform to execute a given program snippet, taking into account the structural properties of the program to offload and metadata describing the environment and conditions in which the execution takes place. These models analyze source code to determine the optimal configuration for tasks on different platforms, considering metrics such as performance or energy. This approach eliminates the need for manual porting and profiling of computational kernels, making it easier for developers to code and deploy their applications. By leveraging deep learning algorithms, which are capable of automatically extracting relevant static features from data, a code classifier is used in this process. Through preliminary source code transformations

Table 4.1: Dataset composition [19]. The first two columns are the number of benchmarks in suite (Benchmarks) and the number of unique kernels in suite (Kernels). In the complete dataset, composed by the tuple Code and Meta-information, each suite has a different number of pairs.

Suite	Version	Benchmarks	Kernels	Samples
amd-sdk	3.0	12	16	16
npb	3.3	7	114	527
nvidia-sdk	4.2	6	12	12
parboil	0.2	6	8	19
polybench	1.0	14	27	27
rodinia	3.1	14	31	31
shoc	1.1.5	12	48	48
Total		71	256	680

and the use of a deep learning network as a language model, important code features are identified for decision-making purposes such as assigning appropriate accelerators to specific portions of the code.

Most recent advancements in deep learning for analyzing source code on different platforms depend greatly on a dataset developed by Cummins et al. [19]. This dataset, summarized in Table 4.1, consists of 256 OpenCL [52] kernels that have been profiled on a Intel Core i7-3820 CPU and two GPU models, namely a AMD Tahiti 7970 and a NVIDIA GTX 970. Alongside the code sequences, additional auxiliary information describes the runtime environment in which the computation is performed. Such information is condensed into two number representing the kernel payload size and the work-group size, an OpenCL platform parameter influencing device parallelism. Since each OpenCL kernel can be executed with multiple combination of payload and work-group size, these supplementary information effectively expand the source code collection into a 680-sample dataset. Typically, these auxiliary inputs are directly integrated into the final classification layer of the network to work alongside output from a deep learning language model that extracts crucial information from the code sequences. A significant drawback of this existing dataset is its limited scale and representativeness; it may not adequately cover all scenarios and applications encountered in real-world situations, potentially restricting accuracy and practicality when training models is based on it.

This chapter introduces two enhancement in the training pipeline for deep-learning models aiming at solving the device mapping problem. First, the distribution of the axuiliary information in the dataset proposed in [19] is analyzed, and a novel pre-processing technique is proposed to standardize such metadata and pre-process them before joining them with the language model into the vector which feed the device mapping classifier. Additionally, the training framework is enhanced to support Siamese networks: a specific type of neural network architecture designed to solve tasks involving similarity analysis, such as face recognition and text matching. Two intuitions support the adoption of Siamese networks. First, training a Siamese network boosts the dataset size since it requires feeding the network with pairs of samples, instead of a single sample. This boosts dataset size since the Siamese counterpart of a dataset composed of N samples is a dataset composed of all the possible pairing of the original N samples. The second intuitions relies on veryfing whether using a loss which forces samples beloing to the same class to have similar representations in the features space is a more effective learning strategy to solve the problem of heterogeneous device mapping. The advantages of this training methodology are showcased through its application on DeepLLVM, a state-of-the-art model for heterogeneous device mapping [3], which analyze a sequence of LLVM-IR tokens through convolutive and pooling layers. Finally, a comparison is made between the accuracy achieved by our proposed modifications with state-of-the-art techniques on heterogeneous device mapping to quantify the improvement.

4.2 Methodology

4.2.1 DeepLLVM network topology

This section outlines the preprocessing pipeline and the network architectures of DeepLLVM [3], which this work assumes as baseline. The analysis of DeepLLVM flow consists of two steps, represented in Figure 4.1: source code preprocessing and code classification. During source code preprocessing, significant syntactic elements are identified and translated into a sequential list of integers. Code classification is then performed using a supervised learning method with a Convolutional Neural Network composed of convolution and pooling layers.

OpenCL kernel pre-processing and LLVM-IR tokenization

To enable machine learning models that work with numerical data, it is necessary to have a preprocessing pipeline that can convert source code into a suitable numerical form as detailed in Section 2.1. The DeepLLVM model takes as input a dataset of computational kernels written using OpenCL [52], which is a high-level framework for writing C programs that can run on heterogeneous platforms with multiple execution engines. The Clang compiler is used to translate OpenCL code to LLVM-IR. This allows the learned language model not to be specific to OpenCL, but potentially applicable to other high-level languages as well. DeepLLVM collects the input source code tokens using a customized procedure tailored for processing LLVM-IR sources. This procedure includes two steps: pre-tokenization and post-tokenization. During the pre-tokenization phase, non-code lines such as empty lines and comments are removed from each kernel. The body of the input kernel is isolated and simplified by reducing LLVM-IR array and vector data types. Furthermore, parentheses, math operators, commas, colons, and assignment symbols are treated as separate tokens in the token stream. Complex data types are also simplified during pre-tokenization by replacing constants with placeholders (e.g., `_float_constant` for real constants), resulting in a significant reduction in code fragment length. Following pre-tokenization, any sequence of characters separated by spaces can be identified as a token. Post-tokenization transformations are applied to the tokens to implement higher-level generalizations, such as removing LLVM-IR meta-data and replacing variable and function names with anonymous placeholders. Additional anonymization operations are described in [3]. The output of the post-tokenization phase is a set of tokens assigned unique numerical identifiers, which will be processed by the language modeling network for classification in downstream tasks.

CNN-based language modelling and downstream classifier

The deep-learning model for device mapping uses a tuple of two inputs: the numerical sequence representing the preprocessed LLVM-IR source and auxiliary data that defines the context of OpenCL kernel usage. DeepLLVM consists of two components: a language model that converts the input token sequence into a multidimensional vector, and a features classifier that analyzes the output of the language model to predict the optimal device for kernel offloading. The network input is a tensor composed of batch-size sequences, each one composed of a token sequence with appropriate padding or trimming. The token indexes are projected into a metric space using an embedding layer, as described in the background section. The Embedding Layer translates the token sequence and projects each element into an embedding space resulting in a list (of length batch-size) of sequences, where each token has been transformed into a vector in the embedding space. The weights of the Embedding Layer determine how each token is mapped to its corresponding embedded vector. During training, the projection in the embedding space initially starts from random conditions. The DeepLLVM language model is inspired by sentence sentiment classification techniques to reduce the output of the embedding layer into a single point in the feature space. The CNN implementation consists of a one-dimensional convolution layer followed by global max-pooling. The convolutional layer applies multiple filters to process each sequence produced by the embedding layer, and then selects the maximum value using global max pooling. Once the language model reduces the original token sequence to a single point in space representing the analyzed kernel, this vector is combined with auxiliary information into a single vector. The resulting vector is then classified using two dense, fully connected layers that output the model's prediction.

4.2.2 Enhanced auxiliary inputs processing pipeline

Meta-information is used to provide the source code analysis with contextual information regarding the environment in which the application runs. For the dataset considered in this study [19] such information is constituted by a tuple of two elements $D_M : \{A_1, A_2\}$ where A_1 represents kernel payload (measured in bytes) which impacts data movement phases, and A_2 denotes work-group size, an OpenCL platform parameter influencing device parallelism. The distribution of the two auxiliary meta-information is shown in Figure 4.2.

To investigate the usage of auxiliary information, the original dataset is modifyed by filtering out source code and metadata pairs with inconsistent labeling. An auxiliary input pair is said to be inconsistently labelled, if its label depends on the specific source code being paired. This results in a separate dataset consisting only of auxiliary inputs. This thesis trains two commonly used classifiers for device mapping analysis - a decision tree and a multi-layer perceptron. The decision tree classifier was previously successful in studies such as [44] and [55], while multi-layer preceptron serves as a baseline for deep learning techniques applied to device mapping. Stratified 10-Fold Cross Validation [53] was employed using both decision tree and multi-layer perceptron models, repeating each experiment ten times with random seeds.

The experimental findings are presented in Table 4.2. The decision tree model exhibits a satisfactory 73% accuracy, which is deemed acceptable given the inclusion of meta-information. In contrast, the MLP model struggles to learn and achieves an approximate accuracy rate of 50%. The challenge lies in the fact that Deep-learning techniques assume properly formatted data with values falling within the range of or [-1, 1]. This requirement ensures accurate gradient computation and prevents issues associated with numerical instabilities during learning. Unfortunately, this condition does not hold for the auxiliary inputs under consideration.

Table 4.2: Results of meta-information dataset classification using decision tree and multi-layers perceptron for raw and normalized data.

	AMD			
	ACC		MCC	
	DT	MLP	DT	MLP
$\widetilde{D}_M$ Raw	0.738	0.465	0.488	-0.012
$\widetilde{D}_M$ Normalized	0.711	0.731	0.422	0.472

	NVIDIA			
	ACC		MCC	
	DT	MLP	DT	MLP
$\widetilde{D}_M$ Raw	0.727	0.535	0.456	0.074
$\widetilde{D}_M$ Normalized	0.724	0.636	0.453	0.248

To address this common problem in device mapping methodology, a preprocessing procedure is implemented on the training data. This procedure consists of three normalization steps: power transformation, standard scaling, and min-max scaling. The power transform applies parametric monotonic transformations to make the distribution of data more Gaussian-like. The standard scaler removes the mean and scales the data to have unit variance. Lastly, the min-max scaler scales the values of the data within a range of [-1, 1], which reduces the feature range that needs to be processed by the multi-layer perceptron algorithm. The findings of this procedure are documented in Table 4.2. Now, both the MLP and decision tree classifiers can effectively learn from both versions of the dataset. It is worth noting that although the performance of these two models is similar, the multi-layer perceptron model holds more potential as it can be easily integrated into a code analysis framework. The proposed techniques offer an improvement to the methodology described by DeepLLVM, as shown in this result. During training, certain scaling parameters are acquired to preprocess auxiliary inputs. Additionally, the network topology is adjusted by incorporating a multi-layer perceptron layer that processes the auxiliary input before merging it with the output of the language model.

4.2.3 Siamese training topology and Contrastive Loss

Siamese networks [14] are a specific type of neural network architecture that are particularly useful for tasks involving similarity analysis, such as face recognition, signature verification, image comparison, and text matching. These networks offer versatility and can be employed in diverse fields where comparing the likeness or dissimilarity between data points plays a critical role. One distinguishing feature of Siamese networks is their incorporation of twin subnetworks with identical architectures and parameters. This design allows them to process pairs of input samples concurrently while learning representations that assign similar features to instances from the same class while separating those with different labels. Through this approach, Siamese networks can effectively uncover and generalize distance metrics based on available data, making them an influential tool across various applications.

Siamese networks encompass a learning approach for training neural networks on similarity tasks, rather than just being limited to network architecture. Figure 4.3 provides an intuition of the working principles of Siamese networks. During the training phase, siamese samples comprise two input samples that undergo processing by twin networks sharing weights. Each twin network independently processes one of the input data points and extracts a feature vector representing the data. The resulting feature vectors from both twin networks are then compared using either a similarity or distance metric. The Siamese network is trained with the objective to optimize a loss function that promotes similarity between feature vectors for similar inputs and dissimilarity for dissimilar inputs. By minimizing this loss, the network becomes capable of discriminating between pairs that are similar versus those that are dissimilar. In this work, the Siamese network is trained using a loss function named "Contrastive Loss". The contrastive loss [35] is a commonly used loss function for training Siamese networks. It encourages the network to learn to discriminate between similar and dissimilar pairs of data points. The loss function is designed to minimize the distance between similar pairs and maximize the distance between dissimilar pairs.

$$loss = \begin{cases} d(S_1, S_2)^2, & \text{if } label(S_1) = label(S_2) \\ \max(0, m - d(S_1, S_2))^2 & \text{otherwise} \end{cases}$$

where m, called "margin" is a crucial hyperparameter that can be adjusted to control the trade-off between correctly classifying similar and dissimilar pairs, depending on the specific requirements of the task. After the completion of training epochs, all the projections of the samples in the training set are gathered and a centroid for each class is calculated by taking an average of their corresponding projections. Ultimately, to test the siamese network, the projections for all points in the test fold are computed and a label is assigned depending on the closer centroid.

4.3 Results

4.3.1 Machine learning models and training hyper-parameters

A series of experiments have been performed to assess the impact of modifying the training framework for preprocessing auxiliary inputs and incorporating siamese training on the accuracy of machine learning models for mapping heterogeneous devices. The current study focuses on evaluating two machine learning models, namely CNN and Siamese. The CNN model is influenced by the DeepLLVM network proposed in [3]. In this approach, a 1D convolutional layer and global max pooling filter are utilized to extract relevant features from a sequence of LLVM-IR tokens. These extracted features are then concatenated with auxiliary inputs and classified using a multi-layer perceptron. To further improve the performance of DeepLLVM, an additional fully connected layer is introduced before combining the auxiliary input features with the output from the language modeling sub-network. Furthermore, Subsection 4.3.2 investigates how preprocessing methods for auxiliary inputs along with an added dense layer impact the classification accuracy of the CNN.

Siamese refers to the CNN described in the previous paragraph, without an activation function in the final layer of the fully-connected classifier. This allows the network to project each dataset sample into a two-dimensional space rather than performing classification tasks. During training, every combination of sample pairs in the train folds undergo separate projection into a 2-dimensional space and update network weights based on the obtained contrastive loss value. As described in Section 4.2.3, samples with different labels are moved apart by the loss function while samples with equal labels are penalized proportionally to their projected distance. Upon completion of training epochs, all projections from train samples are collected. A centroid is then computed for each class using these projections. The Siamese network is tested by projecting all points in the test fold and assigning a label based on proximity to centroids.

Table 4.3 describes the established training framework. For the CNN, a batch size of 32 is used and trained for 70 epochs. The Siamese network is trained using batches of 64 samples and lasted for 15 epochs. Both models utilized the Adam optimizer with a learning rate of 0.001. The CNN model is trained for a larger number of epochs and utilizes a learning-rate scheduler to progressively reduce the learning rate as the model progresses. The learning rate scheduler has three parameters: patience, threshold, and factor. After each training epoch, the training loss is compared to that of previous epochs. If there is no significant improvement

Table 4.3: Training and callbacks hyper-parameters used to train the proposed machine learning models.

Parameters		CNN	Siamese
	Epochs	70	15
	Batch size	32	64
Optimizer	Learning rate	1e-3	1e-3
	Weight decay	5e-4	5e-4
LR scheduler	Factor	5e-1	-
	Threshold	1e-4	-
	Patience	5	-
Contrastive loss	Margin	-	2

beyond the threshold for several epochs (patience), then the optimizer's learning rate is multiplied by a factor. This technique helps prevent gradient descent from getting stuck in local minima and improves performance in deep-learning models.

Evaluation of the experiment outcomes involves two metrics: accuracy and Matthews correlation coefficient. MCC is preferred over accuracy and F1 score for assessing binary classifiers in imbalanced class scenarios [33]. The experiments are repeated 10 times with varying fold splits, resulting in mean (μ), standard deviation (σ), and a 95% confidence interval (CI_{95}).

4.3.2 Impact of auxiliary input preprocessing

The impact of the proposed auxiliary input preprocessing technique detailed in this work is validated by checking the classification performance of the CNN and the Siamese model. To enhance the handling of auxiliary inputs in existing literature, two modifications is purposed in this work: Firstly, a series of scaling techniques is introduced to preprocess the raw auxiliary input values present in the dataset. These techniques include using a power transformer based on the Yeo-Johnson method to make the data distribution more Gaussian-like and applying a standard scaler to remove mean and scale them to unit variance. Secondly, a data scaling within a fixed range of -1 and +1 and Lastly, a one-layer fully connected perceptron in the auxiliary input analysis pathway for enhancing the flexibility of feature space manipulation during training. Four separate experiments - A, B, C, and D - are conducted to assess each modification individually. The details of each experiment can be found in Table 4.4a. Experiment A replicates the findings presented in [3] for the CNN model and serves as a benchmark for evaluating siamese network performance.

Table 4.4: Impact of auxiliary input preprocessing on model accuracy.

	Additional model techniques	
Experiment	input pre-processing	input dense-layer
A	No	No
B	Yes	No
C	No	Yes
D	Yes	Yes

(a) Experiments composition.

		AMD Experiments							
		ACC				MCC			
		A	B	C	D	A	B	C	D
CNN	μ	.853	.882	.868	**.890**	.695	.758	.726	**.775**
	σ	.013	.008	.008	**.006**	.027	.015	.017	**.012**
	CI_{95}	.009	.006	.006	**.004**	.020	.011	.012	**.008**
Siam.	μ	.882	.910	.873	**.917**	.757	.816	.738	**.829**
	σ	**.006**	.008	.009	.007	**.012**	.015	.019	.014
	CI_{95}	**.004**	.005	.007	.005	**.009**	.011	.014	.010

(b) AMD Experiments

		NVIDIA Experiments							
		ACC				MCC			
		A	B	C	D	A	B	C	D
CNN	μ	.823	.843	.830	**.873**	.638	.678	.653	**.767**
	σ	.010	**.008**	.009	.009	.021	**.017**	.017	.018
	CI_{95}	.007	**.006**	.006	.006	.015	**.012**	.012	.013
Siam.	μ	.859	.885	.832	**.888**	.713	.765	.657	**.771**
	σ	.010	**.008**	.010	.009	.021	**.017**	.020	.018
	CI_{95}	.007	**.006**	.007	.006	.015	**.012**	.014	.013

(c) NVIDIA Experiments

Table 4.5: Impact of auxiliary input preprocessing on [19] and [8].

		Test on state-of-the-art models					
		AMD			NVD		
		A	B	Δ	A	B	Δ
DeepT.	μ	.814	**.855**	.041	.805	**.839**	.034
	σ	.020	**.015**	-.005	.008	**.007**	-.007
	CI_{95}	.014	**.007**	-.001	.006	**.005**	-.001
CDFG	μ	.864	**.889**	.025	.814	**.853**	.039
	σ	.010	**.007**	-.003	**.006**	.009	.003
	CI_{95}	.007	**.005**	-.002	**.004**	.006	.002

Table 4.6: Comparison with state-of-the-art methodologies.

		State-of-the-art methodologies		
		AMD	NVIDIA	Mean
DeepTune	[19]	.814	.805	.810
NCC/inst2vec	[5]	.802	.810	.806
CDFG	[8]	.864	.814	.839
DeepLLVM	[3]	.853	.823	.838
CNN	this work	.890	.873	.882
Siamese		**.917**	**.888**	**.903**

(a) State-of-the-art methods were re-evaluated in this work using SKF and Training Set with 9/10 folds.

		State-of-the-art methodologies		
		AMD	NVIDIA	Mean
ProGraML	[18]	.866	.800	.833
CNN	this work	.894	.877	.886
Siamese		**.908**	**.879**	**.894**

(b) CNN and Siamese methods were re-evaluated in this work using SKF and Training Set with 8/10 folds.

The results of the four experiments are presented in Tables 4.4b and 4.4c. Experiment D consistently achieves the highest accuracy and MCC scores for both datasets, indicating that normalizing auxiliary input and adding dense layers are effective strategies for improving classification performance. In particular, the CNN model achieves an accuracy of 88.8% on the AMD dataset, outperforming the baseline from experiment A by 3.7%. For the NVIDIA dataset, our proposed modifications result in a 5 increase in accuracy, reaching 87.3%. Moreover, incorporating Siamese networks into the framework further enhances classification performance with top accuracies of 91.7% for AMD dataset and 88.8% for NVIDIA dataset - demonstrating superior performance compared to alternative methodologies used for source code mapping on heterogeneous platforms.

Furthermore, this study demonstrates the effectiveness of auxiliary input preprocessing in enhancing the performance of existing methodologies for source code device mapping. The impact of this preprocessing pipeline on DeepTune and CDFG is presented in Table 4.5. Our experimental results reveal that incorporating auxiliary input preprocessing leads to a minimum accuracy improvement of 2.5%, with a maximum improvement of 4.1% observed in the case of DeepTune when evaluated using the AMD dataset.

4.3.3 Comparative results

Table 4.6 compares the CNN and the Siamese with other state-of-the-art tools that solve the problem of heterogeneous device mapping. The selection is limited to the methodologies which analyze LLVM-IR, plus DeepTune [19]. This study utilizes Stratified K-Fold Cross Validation to train all of the methodologies under test. Our evaluation is based on the average results from 10 different experiments, each with a unique assignment of samples in the folds. Due to the limited size of the dataset, variations in fold composition can have a significant impact on performance for any given model. To ensure fair comparisons between techniques and machine learning models, a statistics derived from multiple experiments are taken into considerations, where the order of samples within folds are varied. For this purpose, the comparison techniques are replicated in this study and each experiment is performed using an identical setup as our models. Specifically, this thesis uses "Stratified K-Fold Training Set with 9/10 folds - 10 Repetitions - Rebuild Folds" methodology. To ensure a fair comparison between the proposed technique and ProGraML [18], which utilizes a smaller training set (80%), we conducted re-training of both CNN and Siamese models using an analogous dataset splitting configuration. Same as before, a "Stratified K-Fold Training Set is used with the exception of 8/10 folds - 10 Repetitions - Rebuild Folds" for this purpose.

The results shown in Table 4.6 demonstrate a noteworthy increase in accuracy for the proposed methods compared to state-of-the-art techniques. The incorporation of auxiliary input processing techniques, such as normalization and an additional dense layer, enhances the classification accuracy of the baseline model found in [3] from 85.5% to an average of 88.2%. By utilizing these techniques during siamese training, even better performance improvements are achieved, surpassing existing methodologies. Additionally, it is worth noting that both auxiliary input normalization and the Siamese framework can be applied to all methods listed in Table 4.6.

4.4 Conclusions

In this study, two deep-learning classifier approaches are introduced to maximize the potential of a dataset containing source code for heterogeneous device mapping. The first approach involves using metainformation data in combination with two different techniques: a decision tree and a multi-layer perceptron. Our goal is to determine the most effective analysis technique for this type of data. Interestingly, while the decision tree generally performed better, integrating code analysis with a multi-layer perceptron made our methodology more promising. By applying a normalization pipeline, satisfactory results are achieved, even when utilizing the multi-layer perceptron approach. Additionally,this thesis introduces a novel training approach called the Siamese Network. This paradigm utilizes contrastive loss to learn from similarities between samples within the same class. The objective of this technique is to improve classification performance by incorporating a loss function that encourages sample similarity and by expanding the training dataset through paired samples rather than individual ones. By employing a new multi-layer perceptron, normalizing meta information, and utilizing the Siamese Network, satisfactory accuracy rates are achieved in heterogeneous device mapping. Specifically, results show an accuracy of 91.7% for AMD devices and 88.8% for NVIDIA devices on the dataset. For future research directions, an exploration of additional techniques aimed at further improving classification accuracy, will be investigated that may include enhancing the utilized language models and addressing any inconsistencies in dataset labels.

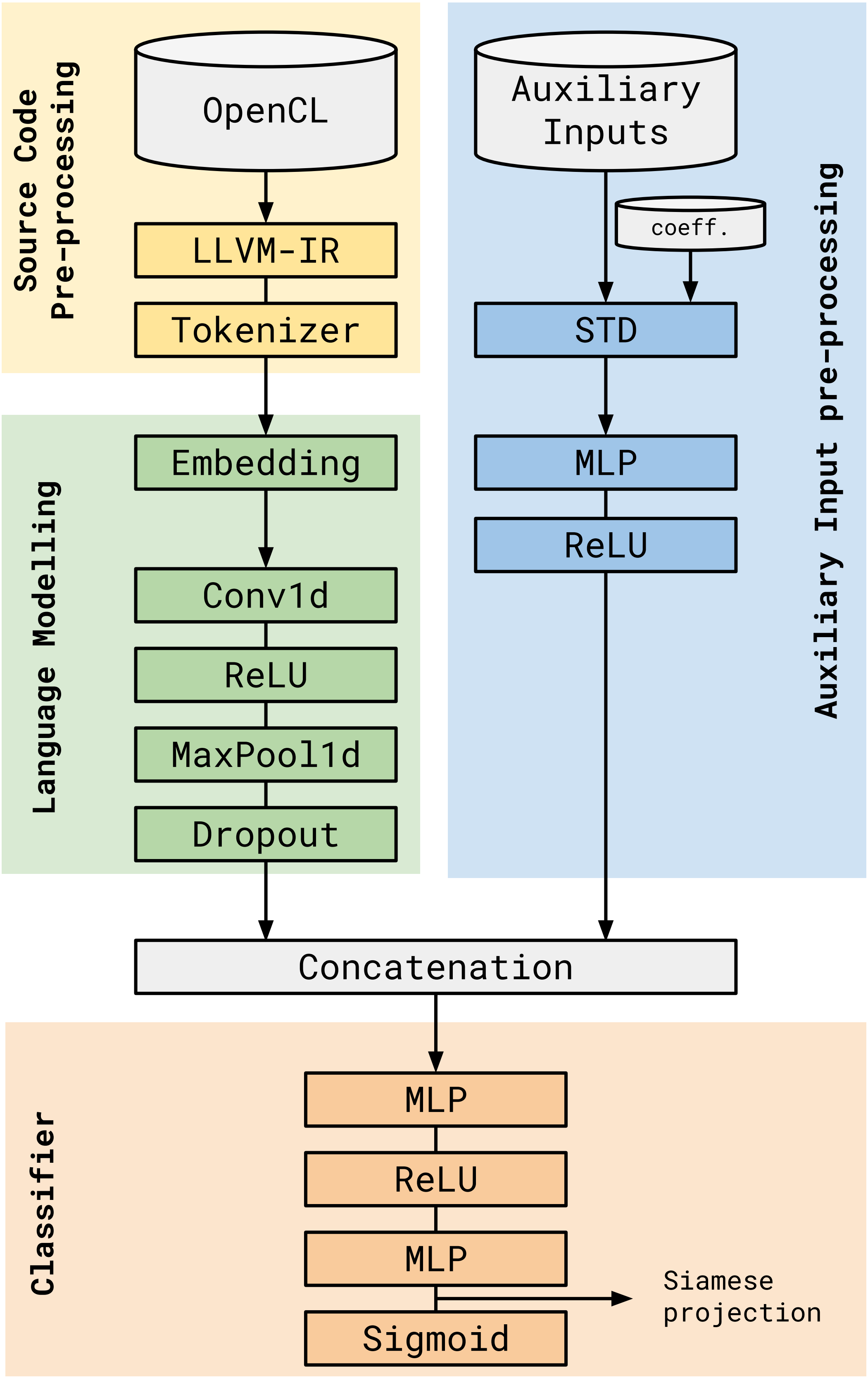


Figure 4.1: Representation of the DeepLLVM [3] data pro-processing and model topology.

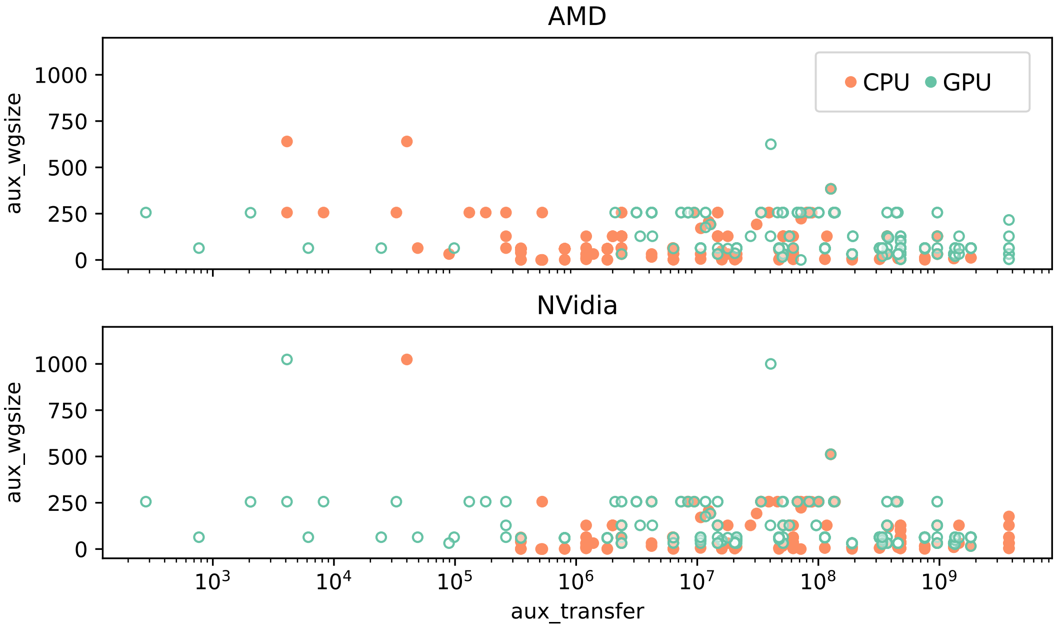


Figure 4.2: Distribution of auxiliary input features for the two datasets considered. Each point is coloured in green or orange depending on its label.

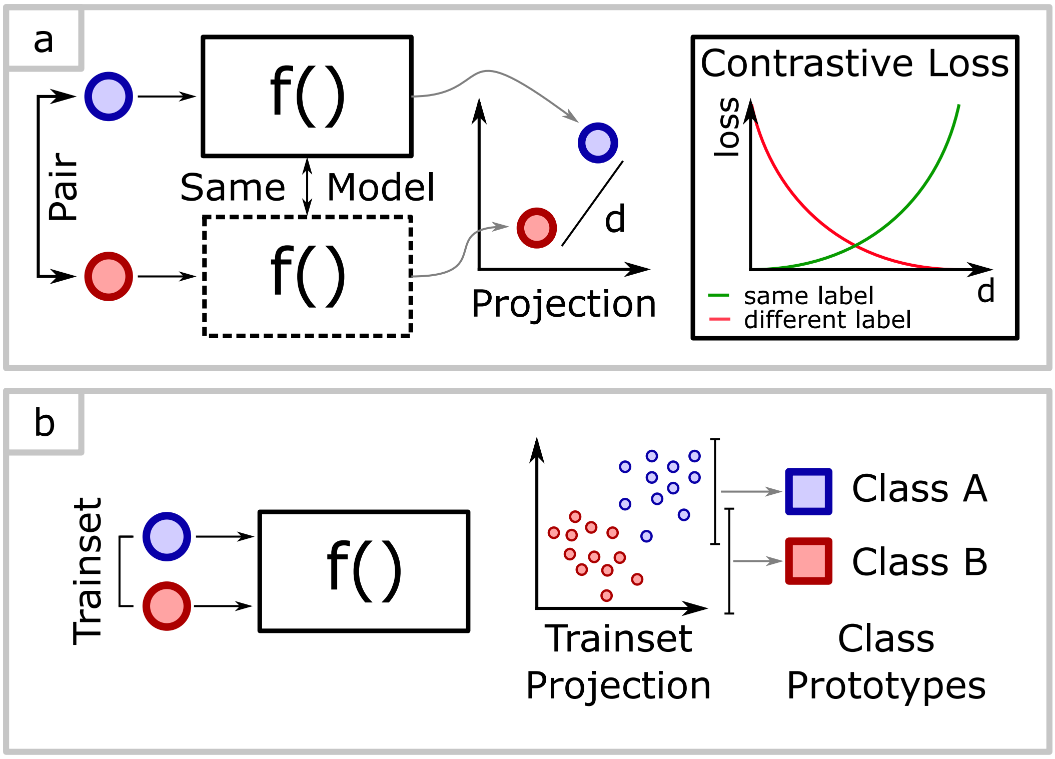


Figure 4.3: Siamese network training. First, the weight of the core network are trainined using the contrastive loss computed on the projections of the points in the train folds. Then, centroids of same class samples are computed and the a label is assigned to each sample in the test set, depending on the closer centroid.

5 Control-Flow Integrity enforcement in the Root-of-Trust

5.1 Introduction

The widespread use of open-hardware platforms in various application domains, including safety and security-critical systems like industrial controllers and autonomous vehicles, has led to extensive research efforts focused on enhancing the security features of these systems. Modern SoC designs are equipped with devices such as Trusted Platform Modules and Root of Trust, on-chip or off-chip IPs featuring cryptographic accelerators, secure storage, and often employing physical measures to ensure robustness against physical attacks, such as power monitoring. These systems are increasingly employed to support secure boot and firmware signature verification, effectively preventing the execution of malicious code.

While secure boot and firmware verification schemes ensure software authenticity at boot time, embedded systems are often programmed using memory-unsafe languages, such as C-derived languages. Such approach has the advantage of granting great freedom to the system developer in terms of memory management, and enable aggressive source code optimization. However, it comes at the cost of a increased risk of bugs and memory corruptions which can potentially introduce security vulnerabilities into the system. Read and write vulnerabilities can be exploited by a determined attacker to bypass traditional memory protection mechanisms [49][17] and alter the control flow of the victim program using Return-Oriented Programming, or other Code-Reuse techniques, to trigger malicious, potentially dangerous, behaviours [46].

To address security threats posed by CRAs, modern SoC designs include Control-Flow Integrity enforcement policies. These policies ensure that an application's control flow follows the constraints established during development and alerts the platform runtime of any violations [1]. In addition to software-based approaches, there are hardware solutions that aim to enhance control flow integrity by extending the system-on-chip architecture with runtime checking capabilities. These hardware solutions can be classified into three main categories: ISA extensions [22][32], hardware monitors [50], and programmable co-processors [23][24]. While implementing ISA extension techniques requires a specific toolchain and designing hardware monitors involves creating a new hardware IP from scratch, security co-processors provide the ability to implement customized policies through software by utilizing a dedicated programmable core that leverages a reserved side-channel for obtaining control-flow information from the main core, paying the area cost of a secondary core which enforce the CFI policy.

This chapter describes an innovative co-processor-based architecture for implementing custom CFI policies on the modern RISC-V platform described in Section 2.2 and is inspired by [15] that includes the OpenTitan RoT and a RV64GC CVA6 host processor [57]. Our approach relies on exploiting the OpenTitan RoT, which is already present on the platform to enable Secure Boot and Remote Attestation, as a CFI co-processor to harness the RV32IMAC Ibex [21] core to execute custom CFI policies in software. This approach eliminates the need for a separate security monitor and optimizes the use of the RoT, which is usually unused after initial setup. It leverages the built-in security features of the RoT, such as private tamper-proof storage access, to enhance security beyond what alternative state-of-the-art solutions offer. The chapter has the following contributions:

Design: Enhancement of the architecture of a RISC-V SoC for autonomous vehicles allow OpenTitan to monitor and track control flow instructions executed by the host processor. Additionally, I improve the commit stage of the host core to filter these instructions and store them in a FIFO until they are deemed "safe" by the monitoring system. Moreover, (iii) the OpenTitan Ibex firmware is extended to analyze the selected instructions and detect any control flow violations.

Exploration: Characterization of the overhead of running CFI checks in the RoT by measuring the runtime overhead of the CFI enforcement scheme on a set of benchmark applications.

Implementation: The effectiveness of the implemented system is demonstrated through the incorporation of a return address protection policy based on a shadow stack. Additionally, an examination and comparison are conducted between our solution's runtime penalty and hardware overhead in relation to the original design.

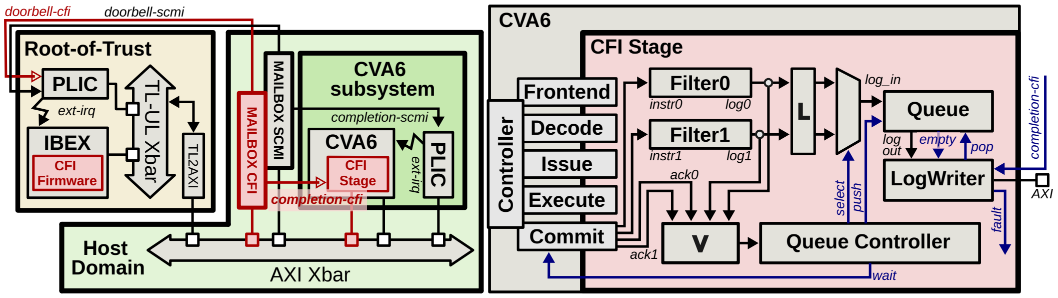


Figure 5.1: Architecture of TitanCFI. The diagram highlights the proposed architectural modification to the SoC (left) and CVA6 core (right)

5.2 CFI Extensions and OpenTitan Firmware

To incorporate the OpenTitan Root-of-Trust as a CFI co-processor, enhancements have been made to the baseline PULP platform described in Section 2.2 and in [15]. As shown in Figure 5.1, these modifications involve implementing a side channel for transmitting control-flow data between the CVA6 host core and OpenTitan. Specifically, changes have been made to expand both the CVA6 commit stage and the SoC communication system. The expansion of the CVA6 commit stage involves several actions: filtering control-flow operations from the retired instructions stream, extracting relevant metadata from the host core scoreboard, and forwarding them to RoT through enqueuing. Alongside hardware adaptations, specialized firmware has also been developed for OpenTitan to enable proper enforcement of Control Flow Integrity. This development ensures compatibility with custom firmware designs while supporting secure data exchange between CVA6 and RoT through an added shared memory region within the SoC communication system.

5.2.1 SoC Modifications and CFI Mailbox

To ensure adherence to the desired control-flow graph, runtime CFI policies require gathering information on the executed control-flow of the target process. This study extracts control-flow metadata from retired instructions stored in the CVA6 commit stage and transmits them to the Root-of-Trust using a mechanism known as the CFI Mailbox. The CFI Mailbox serves as a shared memory area designed for exchanging information between the host core and the Root-of-trust. The CFI Mailbox design bears similarity to the existing SCMI-like mailbox found in the reference SoC [15]. It comprises several shared memory registers used for data sharing, along with two control registers: the doorbell register and the completion register. These control registers are responsible for sending interrupt requests to signal the availability of new data or when a security check has been conducted, thereby notifying both communicating parties. The shared memory space is designed to have enough capacity to accommodate the CFI metadata necessary for representing a single control flow instruction. When new metadata is prepared for retrieval, the enhanced CVA6 commit stage activates the doorbell register to generate an interrupt in the Root-of-Trust. Unlike a typical SCMI-like mailbox, the completion register in this case is not linked to the host domain interrupt controller. Instead, it is directly connected to the commit stage of the CVA6 core. Its purpose is to signal that a previously retired instruction has been verified and checked by the CFI enforcement policy. The result of this policy can be retrieved from the mailbox, indicating that the Root-of-Trust is prepared to proceed with reading the next commit log.

5.2.2 Host Core Modifications

CFI Filters

The commit stage of the CVA6 core has been expanded to include an instruction scanner for all retired instructions from each commit port. This scanning process is responsible for identifying and selecting control flow operations that require checking. These operations, as outlined in Section 2.3, consist of indirect jumps, function returns, and function calls. Indirect jumps, indirect calls, and function returns are types of control-flow changes where the target destination is determined at runtime based on the value stored in a register that may have been tainted by an attacker due to memory corruption. In contrast, direct calls have their target addresses hardcoded in the code binary itself and cannot be compromised assuming the immutability of code memory. However, it remains necessary to collect information about direct calls to enforce backward edge protection mechanisms like shadow stack, aiming at safeguarding against hijacking attempts of a function return address.

This research implements two CFI filters, one for each commit port of the CVA6 core. A CFI Filter receives a scoreboard entry from the commit port, which represents an executed instruction ready for retirement. The purpose of the CFI Filter is to determine if the retired instruction is relevant to Control Flow Integrity and extract necessary metadata, called a commit log, for enforcing CFI. The commit log consists of a 224-bit packet containing four pieces of information: (i) the program counter representing the instruction address, (ii) the uncompressed binary encoding of the current instruction, (iii) the next program counter value, and (iv), the target address. It is important to consider both the next program counter and the target address when dealing with indirect function calls. This is necessary to verify the forward edge by checking the target address of the called function, as well as protect against any attacks on the backward edge when the function returns by storing the next program counter in a shadow stack.

CFI Queue and Queue Controller

The Control Flow Integrity Queue operates as a First-In-First-Out data structure that holds the commit logs obtained from the CFI Filters. To regulate the flow of data into the CFI Queue, the Queue Controller manages the push signal and occasionally inhibits the commit stage, potentially stalling the pipeline, by preventing the core from retiring further instructions. Furthermore, if either multiple commit ports retire control-flow instructions or if there is insufficient space within the CFI Queue, then these conditions trigger inhibition of the commit stage using controls provided by the Queue Control module. This behavior is necessary due to limitations imposed by having only a single entry FIFO for storing commits in internal storage. Due to its simple pipeline and focus on energy efficiency rather than performance, CVA6 typically does not commit multiple control-flow instructions in the same cycle. As a result, implementing the CFI Queue as a standard FIFO should not cause significant performance degradation while maintaining simplicity in design.

CFI Log Writer

The CFI Log Writer module acts as a bridge between the CVA6 core and the CFI Mailbox. It is implemented as a Finite State Machine that retrieves commit logs from the CFI Queue and triggers write transactions on the SoC interconnect to write control-flow metadata to the CFI Mailbox. While idle, the FSM waits for the presence of at least one commit log in the CFI Queue and a ready signal from the CFI Mailbox. Next, it fetches a commit log from the queue and divides it into data chunks with sizes matching those of the data bus width, which is equal to 64 bits in the reference architecture. Then, AXI transactions are initiated to transmit each chunk of data to transfer all sections of a complete commit log into the CFI Mailbox. Finally, the CFI Log Writer triggers an interrupt in the Root-of-Trust by setting the doorbell register and it transits into a "waiting" state, where it remains idle until it receives a completion signal asserted by Root-of-Trust firmware. Once the completion signal is received, the FSM reads the result provided by the CFI enforcement policy from the CFI Mailbox and triggers an exception if any control flow violation is detected.

Finally, the FSM returns to idle and it is ready to pop a new commit log from the CFI Queue.

5.2.3 OpenTitan Firmware Design

In this research, the desired control flow integrity policy is incorporated into the firmware of the OpenTitan Root-of-Trust using C programming language. To achieve this, an additional external interrupt, corresponding to the CFI Mailbox doorbell, is added to the Interrupt Controller of OpenTitan. The CFI policy is then implemented as an interrupt service routine associated with the CFI Mailbox external interrupt. In the given software architecture, it is necessary to implement each CFI policy in three steps: IRQ entry, policy enforcement, and IRQ exit. When the OpenTitan interrupt controller receives a CFI Mailbox interrupt, it wakes up the Ibex microcontroller. Then, Ibex jumps to the CFI interrupt service routine and reads the commit log from the CFI Mailbox. This commit log contains information about the control-flow event that needs to be verified. Next, the binary encoding of the uncompressed RISC-V instructions is used to determine whether the control flow event corresponds to a function call, function return, or indirect jump. Based on this analysis, it applies appropriate programmed CFI enforcement policies. Lastly, the Ibex writes a binary value indicating if any control-flow violation occurred in the CFI Mailbox. It also sets the CFI Mailbox Completion register to signal that checks are complete, the host core can read the outcome of the enforcement policy, and OpenTitan is ready to handle the next commit log.

5.3 Experimental Results

To evaluate the hardware and runtime requirements of the proposed CFI enforcement scheme, we synthesized the modified architecture on an FPGA. We also conducted simulations using two benchmark suites to estimate how these architectural modifications affected hardware utilization and runtime overhead. The synthesis process utilized Xilinx Vivado 2020.2, targeting the Virtex UltraScale+ VCU118 system. The benchmarks used in this study were obtained from EmBench-IoT v1.0 and RISC-V-Tests, which were compiled with a standard RISC-V toolchain including GCC 12.2.0 with optimization level `-O3`.

5.3.1 OpenTitan firmware analysis

To evaluate the performance impact of implementing Control-Flow Integrity enforcement policies in the Root of Trust, a widely known protection policy called shadow-stack is implemented [12]. The implementation proposed in this work analyzes the binary instructions to differentiate between call and return instructions. When a call instruction is encountered, the expected return address is extracted from the commit log and pushed into the shadow stack. If a return instruction is detected, its return address is compared with the value popped from the shadow stack. Any discrepancy is reported as a security violation. Additionally, during both scenarios, the firmware ensures that there are no overflow or underflow issues in the shadow stack. To this purpose, the shadow stack is authenticated when saved or restored from main memory using the HMAC cryptographic accelerator available in the OpenTitan platform.

The cost of implementing the mentioned policy in OpenTitan is presented in Table 5.1. The table categorizes the instructions into *IRQ* and *CFI*, as well as *Logic*, *Memory-RoT*, and *Memory-SoC*. The former category distinguishes instructions involved in IRQ handling, such as spilling registers to the stack and clearing the interrupt pending bit, from those implementing the CFI policy, such as accessing the mailbox or pushing the return address in the stack. The latter category considers the purpose of the executed instruction, separating memory accesses, and distinguishing between RoT private scratchpad and SoC memory, from other operations. Specifically, the *IRQ* section displays the cycle cost for implementing the OpenTitan firmware described in Subsection 5.2.3. The results indicate that the implementation of the CFI policy incurs a significant cost, resulting in approximately 258 to 276 cycles per control-flow operation. However, it is important to note that a considerable portion of this overhead is attributed to the OpenTitan microarchitecture. Upon analyzing the results, two main issues become apparent: a high proportion of cycles dedicated to IRQ handling and added overhead due to memory accesses. Around 60% of the total number of cycles required for implementing the CFI policy are spent on IRQ handling. Also, the examination of simulator-generated traces reveals that there is an average delay of 45 cycles from when the host core commits the instruction which writes the CFI Mailbox doorbell register until the Ibex core awakes. In addition, the overhead of accessing the OpenTitan private scratchpad is about 5 cycles per access and Ibex spills and restores 6 registers during IRQ entry and exit, resulting in a cumulative overhead of 105 cycles regardless of the implemented CFI policy. On the other hand, implementing return address protection in software requires between 48 and 58 opcodes for the CFI policy logic. This demonstrates that software implementation is feasible and may be a more cost-effective option compared to designing custom hardware modules, as long as the RoT architecture is optimized to achieve a high IPC.

The sections of Table 5.1 labeled as *Polling* and *Optimized* present the performance results of the Root-of-Trust firmware after implementing two potential optimizations to address the issues mentioned earlier. In the *Polling* firmware, additional cycles are used for busy waiting by polling the doorbell bit of the CFI Mailbox after verifying an instruction. This strategy aims to reduce the overhead associated with IRQ handling procedures and allows direct execution of CFI enforcement logic when a new control flow instruction is already available in the CFI Queue. This optimization does not require any hardware modifications and achieves an average policy enforcement time of 112 cycles, resulting in approximately 58% savings compared to approaches without busy waiting loops.

A more aggressive optimization, represented by the *Optimized* section of Table 5.1, involves redesigning the interconnect system, and substituting it with a low-latency interconnect. Substituting the internal OpenTitan interconnect with a low-latency interconnect would enable accessing the OpenTitan private memory and peripherals in a single cycle and the SoC memory in approximately 8 cycles, instead of 12. In this last scenario, enforcing return address protection requires 73 cycles on average, more that 70% less than the baseline *IRQ* firmware.

5.3.2 Runtime overhead

To evaluate the impact of CFI enforcement on runtime performance, two suites of benchmarks are run, comparing the overhead resulting from CFI enforcement with the one obtained by alternative solutions in the state-of-the-art. Slowdown is measured by simulating the RTL of a reference SoC and analyzing the cycle-accurate execution trace, which provided us with the number of cycles needed to execute each instruction. These traces were then provided as input to a trace-driven model to estimate the latency introduced by CFI enforcement. Based on the firmware analysis reported in Table 5.1, three different latencies are emulated: 267 cycles for textitIRQ firmware, 112 cycles for *Polling* firmware, and 73 cycles for the *Optimized* variant.

Table 5.2 presents a comparison of the runtime overhead between our approach and two state-of-the-art architectures, namely DExIE [50] and FIXER [22]. It can be observed from the first section of Table 5.2 that our solution incurs lower runtime overhead compared to DExIE [50], taking into account the best obtained results reported in their paper. However, it is important to note that the authors of [50] also mention a reduction in clock frequency when using their security monitor. While compensating for clock reduction makes the overhead of DExIE [50] negligible, Table 5.2 highlights how enforcing CFI in the RoT leads to minimal overhead compared to custom hardware monitor in 3 out of 4 benchmarks tested by [50].

The comparison with FIXER is more complicated because the authors of [22] report a 1.5% runtime overhead for their solution, without showing the breakdown of how such results were obtained. Based on our observations, this work incurs a mean overhead of approximately 5% in most cases, except for the `dhrystone` benchmark. This result is only slightly higher than what was reported by the authors of [22].

Comparing the performance overhead of TitanCFI with alternative CFI enforcement techniques provides valuable insights to compare the performance of alternative approaches to the same problem. However, these comparisons do not offer a comprehensive understanding of the real-world performance degradation that can be expected. Two major concerns arise: the selection of functions with a relatively low number of control-flow instructions and the use of benchmarks that are primarily focused on assessing core performance and are not relevant to the problem of CFI enforcement. To address these limitations, this work takes a different approach. Instead of focusing on limited function sets or non-relevant benchmarks, we assess the overhead of CFI enforcement in the Root-of-Trust across a larger pool comprising the EmBench-IoT suite and most RISC-V-Tests. This allows for a more robust evaluation in terms of both benchmark diversity and scalability to real-world scenarios. Table 5.3 presents the execution cycle count, number of retired control flow instructions, and slowdown achieved by this work with a CFI queue size of 8 for each benchmark. While the implementation proposed here currently incurs some overhead and requires optimization to support benchmarks with higher control flow instruction counts, it generally results in less than 10% overhead for most tested kernels.

To improve the performance of this work, two main approaches can be taken. Firstly, ad-hoc static source code analysis techniques should be developed to identify hotspots in the code to perform a selective function call inlining to reduce the burden on the CFI enforcement module. Additionally, supporting per-thread CFI enforcement would provide selective protection for processes exposed at the system boundary when dealing with potentially tainted data and inputs. It should also be noted that the benchmarks in Table 5.3 which impose the highest overhead are not kernel that requires CFI enforcement, since they aim at performing linear algebra or physics computation. In real-world scenarios, these benchmarks wouldn't typically need protection against control-flow attacks.

5.3.3 Hardware utilization overhead

The hardware overhead of the proposed architectural modifications is measured conducting FPGA synthesis to determine the utilization of LUTs, registers, and BRAMs with respect to the original design. The results are summarized in Table 5.4, which demonstrate that the proposed modifications have a minimal effect on hardware resources. Specifically, they account for less than 1% of the entire SoC's resource utilization and less than 6% when considering only the host core. Comparing these findings with those reported by [50], it is evident that this solution outperforms theirs in terms of resource efficiency. The described implementation utilizes approximately 60% fewer LUTs, 2% fewer registers, and does not require any BRAM slice usage. Importantly, these architectural changes do not compromise the maximum operating frequency of the SoC either.

5.4 Security Assumptions and Implications

This research aims to safeguard software developed in a memory-unsafe language, like C, which is susceptible to bugs and memory vulnerabilities that may be exploited. The protection scheme implemented in this work assumes that other entities within the System-on-Chip cannot tamper with the CFI Mailbox. This assumption is reasonable because additional security components, such as RISC-V Physical Memory Protection, can be configured to prevent access to specific memory regions. As a result, any attempts to read from or write into these protected areas will trigger an access fault exception. Furthermore, it is assumed that only the Ibex microcontroller has secure and exclusive access rights to the private memory of the Root of Trust. No other party should have visibility or be able to modify data in this portion of memory.

Enforcing control-flow integrity policies within the Root-of-Trust allows for the utilization of its private memory and hardware accelerators, further enhancing security guarantees. One example is utilizing the internal RoT scratchpad to securely store sensitive information like the shadow stack. Unlike many CFI architectures that rely on reserved pages in virtual memory protected by the operating system, implementing CFI within the RoT ensures greater security as its private memory remains inaccessible to any entity in the SoC. In a multi-process scenario where numerous processes require protection, it is improbable for all the necessary CFI metadata to be stored in the Root-of-Trust private memory simultaneously. As a result, transferring data to the SoC main memory occasionally becomes necessary.

Many CFI architectures in the state-of-the-art [23, 24] store CFI data in reserved pages of virtual memory protected by the Operating System, assuming the Operating System is trusted and secure. One potential solution involves statically allocating a specific region of main memory for the Root-of-Trust using technologies like RISC-V PMP. Alternatively, having access to the cryptographic accelerators provided by the Root-of-Trust enables the authentication of CFI metadata, as suggested by ZipperStack [38], before storing it in unsecured locations within main memory.

5.5 Conclusions

This study explores the feasibility of implementing Control-Flow Integrity enforcement policies in software within a Root of Trust environment. By leveraging tamper-proof memory and cryptographic accelerators, it is possible to enhance the security guarantees offered by the CFI enforcement scheme without requiring custom hardware monitors or modifications to the host core pipeline. Additionally, it describes how to leverage the attributes of Root-of-Trust, such as secure memory and cryptographic accelerators, to strengthen the security assurances offered by the CFI enforcement mechanism. The proposed solution is showcased by enhancing the design of the RISC-V SoC described in Section 2.2. The approach demonstrates minimal hardware overhead and maintains timing integrity. Through tests conducted on EmBench-IoT and RISC-V-Tests benchmark suites, it was observed that the majority of benchmarks did not impose any or less than 10% runtime overhead. Comparatively, the impact of the proposed CFI enforcement scheme on system performance is similar to alternative state-of-the-art architectures [50] [22] for most benchmarks. Further research will focus on evaluating our approach with different RoTs, addressing challenges specific to OpenTitan, and implementing our protection scheme on more advanced platforms such as multi-core hosts with varying CFI policies.

Table 5.1: Cycles required to implement the return address protection policy in OpenTitan

Op.			Instructions [#]			Cycles [#]			Cycles [%]		
			IRQ	CFI	TOT	IRQ	CFI	TOT	IRQ	CFI	TOT
IRQ	CALL	Logic	8	15	23	59	27	86	23	10	33
		Mem. RoT	14	5	19	74	28	102	29	9	38
		Mem. SoC	2	4	6	22	48	70	10	19	29
		TOT	24	24	48	155	103	258	62	38	100
	RET.	Logic	8	15	33	59	45	104	21	16	37
		Mem. RoT	14	5	19	74	28	102	27	10	37
		Mem. SoC	2	4	6	22	48	70	9	17	26
		TOT	24	34	58	155	121	276	57	43	100
Polling	CALL	Logic	–	15	15	–	27	27	–	26	26
		Mem. RoT	–	5	5	–	28	28	–	27	27
		Mem. SoC	–	4	4	–	48	48	–	47	47
		TOT	–	24	24	–	103	103	–	100	100
	RET.	Logic	–	25	25	–	45	45	–	37	37
		Mem. RoT	–	5	5	–	28	28	–	23	23
		Mem. SoC	–	4	4	–	48	48	–	40	40
		TOT	–	34	34	–	121	121	–	100	100
Optimized	CALL	Logic	–	15	15	–	27	27	–	42	42
		Mem. RoT	–	5	5	–	5	5	–	08	08
		Mem. SoC	–	4	4	–	32	32	–	50	50
		TOT	–	24	24	–	64	64	–	100	100
	RET.	Logic	–	25	25	–	45	45	–	55	55
		Mem. RoT	–	5	5	–	5	5	–	06	06
		Mem. SoC	–	4	4	–	32	32	–	39	39
		TOT	–	34	34	–	82	82	–	100	100

Table 5.2: Slowdown compared to [50] and [22]

	Benchmark	Slowdown [%]				
		[50]	[22]	Opt.	Poll.	IRQ
EmBench	aha-mont64	48	n.a.	–	–	–
	edn	47	n.a.	1	1	2
	matmult-int	48	n.a.	–	–	1
	ud	48	n.a.	12	18	43
RISC-V Tests	rsort	n.a.		–	–	1
	median	n.a.		3	5	12
	qsort	n.a.	2	–	–	1
	multiply	n.a.		2	3	6
	dhrystone	n.a.		360	553	1318

Table 5.3: Analysis of slowfown for EmBench-IoT and RISC-V Tests.

	Benchmark	Cycles	CF	Slowdown [%]		
				Opt.	Poll.	IRQ
EmBench	aha-mont64	2.51E+6	1.50E+1	–	–	–
	crc32	3.49E+6	1.50E+1	–	–	–
	cubic	1.10E+6	2.01E+4	46	107	390
	edn	4.23E+6	3.67E+2	–	–	–
	huffbench	3.49E+6	2.28E+3	1	3	11
	matmult-int	4.69E+6	2.05E+2	–	–	–
	minver	4.75E+5	4.50E+3	–	7	153
	nbody	1.21E+5	4.29E+3	163	301	849
	nettle-aes	5.20E+6	7.95E+2	–	–	–
	nettle-sha256	4.73E+6	8.57E+3	1	2	11
	nsichneu	5.24E+6	1.70E+1	–	–	–
	picojpeg	4.97E+6	2.14E+4	5	15	58
	qrduino	4.61E+6	4.35E+3	–	–	–
	sglib-combined	3.67E+6	2.62E+4	9	32	142
	slre	3.57E+6	6.69E+4	38	110	401
	st	1.47E+5	2.31E+2	–	–	2
	statemate	3.22E+6	2.75E+4	–	–	129
	ud	1.87E+6	2.98E+3	–	–	–
	wikisort	4.38E+5	7.69E+3	94	158	418
RISC-V Tests	dhrystone	4.57E+5	2.25E+4	260	452	1215
	median	2.53E+4	1.10E+1	–	–	–
	memcpy	1.20E+5	1.10E+1	–	–	–
	mm	1.41E+6	2.33E+5	1108	1752	4311
	mt-matmul	5.76E+4	2.38E+2	11	22	65
	mt-memcpy	4.08E+5	1.80E+1	–	–	–
	mt-vvadd	1.48E+5	3.30E+1	–	–	–
	multiply	3.72E+4	9.00E+0	–	–	–
	pmp	9.01E+5	5.90E+1	–	–	–
	qsort	2.68E+5	1.10E+1	–	–	–
	rsort	3.32E+5	1.10E+1	–	–	–
	spmv	1.67E+5	1.10E+1	–	–	–
	towers	2.01E+4	9.00E+0	–	–	–

Table 5.4: FPGA resource utilization compared to [50]

		w.o CFI	CFI	Δ	Overhead
Host	LUT	5.02E+4	5.14E+4	1.16E+3	+2.3 %
	Registers	3.04E+4	3.22E+4	1.77E+3	+5.8 %
	BRAM	6.60E+1	6.60E+1	–	–
SoC	LUT	4.41E+5	4.41E+5	1.33E+3	+0.3 %
	Registers	2.57E+5	2.58E+5	2.19E+3	+0.9 %
	BRAM	2.68E+2	2.68E+2	–	–
[50]	LUT	4.66E+3	8.02E+3	3.36E+3	+72.1 %
	Registers	3.09E+3	5.33E+3	2.24E+3	+72.5 %
	BRAM	1.36E+2	1.42E+2	6.00E+0	+4.4 %

Conclusions 6

Embedded systems are becoming more prevalent across various domains and have become an essential part of our daily lives, improving efficiency in numerous ways. The integration of advanced hardware architectures with sensors and actuators has led to the development of Cyber-Physical Systems which play a crucial role in enhancing everyday tasks by enabling complex monitoring and control functions across industries like manufacturing, energy, healthcare, and transportation. Throughout this research, I focus on creating novel approaches to address the numerous challenges involved in programming contemporary Embedded Systems. This encompasses optimizing platform configurations and task mapping for optimal performance and developing architectural modifications to enforce security measures.

Machine Learning-based Device Configuration

In the context of automatic platform configuration for optimizing performance, a machine learning model was developed to predict the optimal number of cores for minimizing energy consumption in OpenMP kernels on deeply embedded architectures. The focus is on PULP, an advanced parallel ultra-low-power embedded microcontroller with 8 RISC-V cores. Results show that by training the model with static features extracted from source code, it could predict the optimal platform configuration within a 5% tolerance on energy minimization with an accuracy close to 80%. Additionally, incorporating dynamic features into the prediction model only results in approximately a 10% loss in performance, suggesting that static source code analysis is a viable approach to platform optimization.

Deep Learning-based Heterogeneous Device Mapping

In this research, I face the challenge of mapping tasks to heterogeneous devices to optimize runtime performance. To enhance the performance of existing deep learning-based solutions for device mapping, I propose two approaches based on state-of-the-art techniques. The first method presents a pre-processing pipeline for auxiliary input data, which enhances classification performance by standardizing the data during training and incorporating a multi-layer perceptron layer in the deep-learning model to reproject the auxiliary inputs. This is later concatenated with metadata and language modeling output. This technique has been implemented on three state-of-the-art models to demonstrate its effectiveness in improving classification performance. Additionally, I implement a novel technique called the Siamese Network to improve the training framework for device mapping. This approach utilizes contrastive loss to enhance classification accuracy by emphasizing sample similarities with the same label. By incorporating paired samples instead of individual ones, the training dataset is expanded, leading to improved results. The proposed improvements were applied to DeepLLVM, which is a state-of-the-art model for heterogeneous device mapping. The evaluation results demonstrat that our approach achieved an accuracy of 91.7% for AMD devices and 88.8% for NVIDIA devices on the dataset used in this study. These accuracies surpassed those obtained by alternative state-of-the-art models, confirming the effectiveness of our proposed techniques.

Control-Flow Integrity enforcement in the Root-of-Trust

In this research, the focus is on enforcing the security guarantee offered by embedded RISC-V platform by implementing Control-Flow Integrity enforcement policies in software within a Root of Trust environment. The goal is to enhance the security measures provided by the CFI scheme without the need for custom hardware monitors or extensive modifications to the host core pipeline. Moreover, I discuss the advantages of leveraging features like tamper-proof memory and cryptographic accelerators that lead to further strengthening the security guarantees of CFI in conjunction with Root-of-Trust attributes. The proposed approach showcases limited hardware impact and preserves timing integrity. Testing conducted on EmBench-IoT and RISC-V-Tests, two well-known bare-metal benchmark suites, indicated that the majority of benchmarks imposed either no overhead or less than a 10% increase in runtime. In comparison to other current architectures, the effect on system performance from implementing the suggested CFI enforcement scheme is similar for most benchmarks.

Bibliography

[1]M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. “Control-Flow Integrity Principles, Implementations, and Applications”. In: *ACM Trans. Inf. Syst. Secur.* (2009). DOI: 10.1145/1609956.1609960 (cit. on pp. 21, 51).

[2]F. Barchi, E. Parisi, A. Bartolini, and A. Acquaviva. “Deep Learning Approaches to Source Code Analysis for Optimization of Heterogeneous Systems: Recent Results, Challenges and Opportunities”. In: *Journal of Low Power Electronics and Applications* 12.3 (2022). DOI: 10.3390/jlpea12030037 (cit. on pp. 5, 8).

[3]F. Barchi, E. Parisi, G. Urgese, E. Ficarra, and A. Acquaviva. “Exploration of Convolutional Neural Network models for source code classification”. In: *Engineering Applications of Artificial Intelligence* 97 (2021), p. 104075. DOI: https://doi.org/10.1016/j.engappai.2020.104075 (cit. on pp. 8, 12, 35, 37, 38, 42, 43, 45, 47, 48).

[4]F. Barchi, G. Urgese, E. Macii, and A. Acquaviva. “Code Mapping in Heterogeneous Platforms Using Deep Learning and LLVM-IR”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC ’19. Las Vegas, NV, USA: Association for Computing Machinery, 2019. DOI: 10.1145/3316781.3317789 (cit. on pp. 6–8, 12).

[5]T. Ben-Nun, A. S. Jakobovits, and T. Hoefler. *Neural Code Comprehension: A Learnable Representation of Code Semantics*. 2018. arXiv: 1806.07336 [cs.LG] (cit. on pp. 6, 12, 13, 45).

[6]T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. “Jump-Oriented Programming: A New Class of Code-Reuse Attack”. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS ’11. Hong Kong, China: Association for Computing Machinery, 2011, pp. 30–40. DOI: 10.1145/1966913.1966919 (cit. on p. 20).

[7]A. Bordes, N. Usunier, A. Garcia-Durán, J. Weston, and O. Yakhnenko. “Translating Embeddings for Modeling Multi-Relational Data”. In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*. NIPS’13. Lake Tahoe, Nevada: Curran Associates Inc., 2013, pp. 2787–2795 (cit. on p. 14).

[8]A. Brauckmann, A. Goens, S. Ertel, and J. Castrillon. “Compiler-Based Graph Representations for Deep Learning Models of Code”. In: *Proceedings of the 29th International Conference on Compiler Construction*. CC 2020. San Diego, CA, USA: Association for Computing Machinery, 2020, pp. 201–211. DOI: 10.1145/3377555.3377894 (cit. on pp. 9, 35, 45).

[9]M. Brohet and F. Regazzoni. “A Survey on Thwarting Memory Corruption in RISC-V”. In: *ACM Comput. Surv.* 56.2 (Sept. 2023). DOI: 10.1145/3604906 (cit. on p. 19).

[10] N. Bruschi, G. Haugou, G. Tagliavini, F. Conti, L. Benini, and D. Rossi. In: IEEE, Oct. 2021. DOI: 10.1109/iccd53106.2021.00071 (cit. on pp. 24, 29, 33).

[11] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. "Control-Flow Integrity: Precision, Security, and Performance". In: *ACM Comput. Surv.* 50.1 (Apr. 2017). DOI: 10.1145/3054924 (cit. on p. 19).

[12] N. Burow, X. Zhang, and M. Payer. "SoK: Shining Light on Shadow Stacks". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00076 (cit. on p. 57).

[13] L. Chen, S. Sultana, and R. Sahita. "HeNet: A Deep Learning Approach on Intel® Processor Trace for Effective Exploit Detection". In: May 2018, pp. 109–115. DOI: 10.1109/SPW.2018.00025 (cit. on p. 21).

[14] X. Chen and K. He. *Exploring Simple Siamese Representation Learning*. 2020. arXiv: 2011.10566 [cs.CV] (cit. on p. 40).

[15] M. Ciani, S. Bonato, R. Psiakis, A. Garofalo, L. Valente, S. Sugumar, A. Giusti, D. Rossi, and D. Palossi. "Cyber Security aboard Micro Aerial Vehicles: An OpenTitan-based Visual Communication Use Case". In: *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2023. DOI: 10.1109/ISCAS46773.2023.10181732 (cit. on pp. 17, 52, 53).

[16] lowRISC CIC. *OpenTitan Official Documentation*. https://opentitan.org/book/doc/introduction.html. 2019 (cit. on p. 17).

[17] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In: *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. USENIX Association, 1998 (cit. on pp. 20, 51).

[18] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather. *ProGraML: Graph-based Deep Learning for Program Optimization and Analysis*. 2020. arXiv: 2003.10536 [cs.LG] (cit. on pp. 9, 10, 35, 45, 46).

[19] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. "End-to-End Deep Learning of Optimization Heuristics". In: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2017, pp. 219–232. DOI: 10.1109/PACT.2017.24 (cit. on pp. 8, 12, 35, 36, 39, 45, 46).

[20] L. Dagum and R. Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55. DOI: 10.1109/99.660313 (cit. on p. 25).

[21] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini. "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications". In: *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2017. DOI: 10.1109/PATMOS.2017.8106976 (cit. on pp. 17, 52).

[22]A. De, A. Basu, S. Ghosh, and T. Jaeger. “FIXER: Flow Integrity Extensions for Embedded RISC-V”. In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2019. DOI: 10.23919/DATE.2019.8714980 (cit. on pp. 22, 52, 58, 59, 61, 63).

[23]L. Delshadtehrani, S. Canakci, B. Zhou, S. Eldridge, A. Joshi, and M. Egele. “PHMon: A Programmable Hardware Monitor and Its Security Use Cases”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020 (cit. on pp. 22, 52, 61).

[24]L. Delshadtehrani, S. Eldridge, S. Canakci, M. Egele, and A. Joshi. “Nile: A Programmable Monitoring Coprocessor”. In: *IEEE Computer Architecture Letters* (2018). DOI: 10.1109/LCA.2017.2784416 (cit. on pp. 22, 52, 61).

[25]H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. “Dark silicon and the end of multicore scaling”. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 365–376 (cit. on p. 35).

[26]R.-V. Foundation. *The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2*. https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf. 2019 (cit. on p. 21).

[27]A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini. “PULP-NN: accelerating quantized neural networks on parallel ultra-low-power RISC-V processors”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378.2164 (Dec. 2019), p. 20190155. DOI: 10.1098/rsta.2019.0155 (cit. on p. 18).

[28]M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini. “Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), pp. 2700–2713. DOI: 10.1109/TVLSI.2017.2654506 (cit. on p. 18).

[29]F. Glaser, G. Tagliavini, D. Rossi, G. Haugou, Q. Huang, and L. Benini. “Energy-Efficient Hardware-Accelerated Synchronization for Shared-L1-Memory Multiprocessor Clusters”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2021), pp. 633–648. DOI: 10.1109/TPDS.2020.3028691 (cit. on p. 19).

[30]D. Grewe, Z. Wang, and M. F. P. O’Boyle. “Portable mapping of data parallel programs to OpenCL for heterogeneous systems”. In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2013, pp. 1–10. DOI: 10.1109/CGO.2013.6494993 (cit. on pp. 11, 27, 28).

[31]D. J. Hemanth. “Automated feature extraction in deep learning models: A boon or a bane?” In: *2021 8th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*. 2021, pp. 3–3. DOI: 10.23919/EECSI53397.2021.9624287 (cit. on p. 11).

[32]*RISC-V Zisslpcfi ISA extension for Control-Flow Integrity*. 2023 (cit. on p. 52).

[33]G. Jurman, S. Riccadonna, and C. Furlanello. “A comparison of MCC and CEN error measures in multi-class prediction”. In: *PloS one* 7.8 (2012), e41882 (cit. on p. 43).

[34]D. P. Khatri, G. Song, and T. Zhu. *Heterogeneous Computing Systems*. 2022. arXiv: 2212.14418 [eess.SY] (cit. on p. 35).

[35] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan. *Supervised Contrastive Learning*. 2021. arXiv: 2004.11362 [cs.LG] (cit. on p. 41).

[36] A. Kurth, W. Ronninger, T. Benz, M. Cavalcante, F. Schuiki, F. Zaruba, and L. Benini. "An Open-Source Platform for High-Performance Non-Coherent On-Chip Communication". In: *IEEE Transactions on Computers* (2021), pp. 1–1. DOI: 10.1109/tc.2021.3107726 (cit. on p. 16).

[37] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665 (cit. on pp. 5, 7, 23, 28).

[38] J. Li, L. Chen, Q. Xu, L. Tian, G. Shi, K. Chen, and D. Meng. "Zipper Stack: Shadow Stacks Without Shadow". In: *Computer Security – ESORICS 2020*. 2020 (cit. on p. 61).

[39] A. Magni, C. Dubach, M. O'Boyle, and B. Pizzi. "Automatic Optimization of Thread-Coarsening for Graphics Processors". In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. 2014 (cit. on p. 23).

[40] J. Merrill. "Generic and gimple: A new tree represen-tation for entire functions". In: 2003 (cit. on p. 5).

[41] T. Mikolov, K. Chen, G. Corrado, and J. Dean. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL] (cit. on p. 13).

[42] F. Montagna, S. Mach, S. Benatti, A. Garofalo, G. Ottavi, L. Benini, D. Rossi, and G. Tagliavini. "A Low-Power Transprecision Floating-Point Cluster for Efficient Near-Sensor Data Analytics". In: *IEEE Transactions on Parallel and Distributed Systems* 33.5 (2022), pp. 1038–1053. DOI: 10.1109/TPDS.2021.3101764 (cit. on p. 18).

[43] A. One. "Smashing the Stack for Fun and Profit". In: *Phrack* 7.49 (Nov. 1996) (cit. on p. 19).

[44] E. Parisi, F. Barchi, A. Bartolini, G. Tagliavini, and A. Acquaviva. "Source Code Classification for Energy Efficiency in Parallel Ultra Low-Power Microcontrollers". In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 878–883. DOI: 10.23919/DATE51398.2021.9474085 (cit. on p. 39).

[45] M. Payer, A. Barresi, and T. R. Gross. "Lockdown: Dynamic control-flow integrity". In: *arXiv preprint arXiv:1407.0549* (2014) (cit. on p. 22).

[46] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. "Return-Oriented Programming: Systems, Languages, and Applications". In: *ACM Trans. Inf. Syst. Secur.* (Mar. 2012). DOI: 10.1145/2133375.2133377 (cit. on pp. 20, 51).

[47] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini. "PULP: A parallel ultra low power platform for next generation IoT applications". In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. 2015 (cit. on pp. 18, 23, 25, 30).

[48]D. Rossi, I. Loi, F. Conti, G. Tagliavini, A. Pullini, and A. Marongiu. “Energy efficient parallel computing on the PULP platform with support for OpenMP”. In: *2014 IEEE 28th Convention of Electrical & Electronics Engineers in Israel (IEEEI)*. 2014, pp. 1–5. DOI: 10.1109/EEEI.2014.7005803 (cit. on p. 24).

[49]K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. “Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization”. In: *2013 IEEE Symposium on Security and Privacy*. 2013. DOI: 10.1109/SP.2013.45 (cit. on p. 51).

[50]C. Spang, Y. Lavan, M. Hartmann, F. Meisel, and A. Koch. “DExIE - An IoT-Class Hardware Monitor for Real-Time Fine-Grained Control-Flow Integrity”. In: *Journal of Signal Processing Systems* (2022). DOI: https://doi.org/10.1007/s11265-021-01732-5 (cit. on pp. 22, 52, 58–61, 63, 65).

[51]N. Stojanovski, M. Gusev, D. Gligoroski, and S. Knapskog. “Bypassing Data Execution Prevention on MicrosoftWindows XP SP2”. In: *The Second International Conference on Availability, Reliability and Security (ARES'07)*. 2007, pp. 1222–1226. DOI: 10.1109/ARES.2007.54 (cit. on p. 20).

[52]J. E. Stone, D. Gohara, and G. Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science & Engineering* 12.3 (2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69 (cit. on pp. 11, 27, 36, 37).

[53]S. Szeghalmy and A. Fazekas. “A Comparative Study of the Use of Stratified Cross-Validation and Distribution-Balanced Stratified Cross-Validation in Imbalanced Learning”. In: *Sensors* 23.4 (2023). DOI: 10.3390/s23042333 (cit. on p. 39).

[54]L. Szekeres, M. Payer, T. Wei, and D. Song. “SoK: Eternal War in Memory”. In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 48–62. DOI: 10.1109/SP.2013.13 (cit. on p. 20).

[55]S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. N. Srikant. “IR2VEC: LLVM IR Based Scalable Program Embeddings”. In: *ACM Trans. Archit. Code Optim.* 17.4 (Dec. 2020). DOI: 10.1145/3418463 (cit. on pp. 12–14, 39).

[56]Z. Wang, G. Tournavitis, B. Franke, and M. F. P. O'boyle. “Integrating Profile-Driven Parallelism Detection and Machine-Learning-Based Mapping”. In: *ACM Trans. Archit. Code Optim.* (Feb. 2014) (cit. on p. 23).

[57]F. Zaruba and L. Benini. “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (Nov. 2019). DOI: 10.1109/tvlsi.2019.2926114 (cit. on pp. 15, 52).

[58]J. Zhang, W. Chen, and Y. Niu. *DeepCheck: A Non-intrusive Control-flow Integrity Checking based on Deep Learning*. 2019. arXiv: 1905.01858 [cs.CR] (cit. on p. 21).

[59]M. Zhang and R. Sekar. “Control Flow Integrity for COTS Binaries”. In: *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 337–352 (cit. on p. 21).