Understanding Code Similarity across Instruction Set Architectures: An Empirical Study

HAONAN YU[†], Institute of Software at Chinese Academy of Sciences, China
JIAXIN ZHU[*†‡], Institute of Software at Chinese Academy of Sciences, China
YINGYING ZHENG[†], Institute of Software at Chinese Academy of Sciences, China
YUWEI ZHANG[†], Institute of Software at Chinese Academy of Sciences, China
WEI WANG[†‡], Institute of Software at Chinese Academy of Sciences, China
JUN WEI[†‡], Institute of Software at Chinese Academy of Sciences, China
TAO HUANG[†], Institute of Software at Chinese Academy of Sciences, China

Software interacts with hardware through Instruction Set Architectures (ISAs), such as x86, ARM, and RISC-V. Although many developers may be unaware of ISA heterogeneity, ISA-specific code is pervasive in foundational software systems that underpin the digital infrastructure of human society. Maintaining separate implementations is common when supporting multiple ISAs in such a foundational software project. This may introduce substantial additional effort. Meanwhile, separate ISA-specific implementations frequently exhibit code similarities across ISAs. While prior code similarity research has largely focused on general-purpose clones or cross-language settings, similarity in ISA-specific implementations remains underexplored. To understand ISA-specific code and their similarities, and to gain insights for better management, we conducted an empirical study of 20 open-source foundational projects that support multiple ISAs. We confirmed the need for separate ISA-specific implementations by identifying the roles and characteristics of large-scale ISA-specific code, with assistance from large language models (LLMs). Our analysis of the ISA-specific code revealed a weighted average similarity of 21.7% across ISAs. We also observed cross-ISA co-change and cross-ISA participation patterns in the development and maintenance of ISA-specific code. By centering on ISA-specific implementations rather than general-purpose clones, this study provides a dedicated empirical characterization of a practically important but underexplored code-similarity setting, yielding evidence that can inform both researchers and practitioners working on ISA-related software engineering.

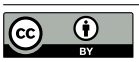
CCS Concepts: • **Software and its engineering → Software creation and management**.

Additional Key Words and Phrases: instruction set architecture, code similarity, co-change, RISC-V.

ACM Reference Format:
Haonan Yu, Jiaxin Zhu, Yingying Zheng, Yuwei Zhang, Wei Wang, Jun Wei, and Tao Huang. 2026. Understanding Code Similarity across Instruction Set Architectures: An Empirical Study. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE112 (July 2026), 24 pages. https://doi.org/10.1145/3808119

[*]Jiaxin Zhu is the corresponding author.
[†]Affiliated with University of Chinese Academy of Sciences (CAS) and Key Laboratory of System Software CAS.
[‡]Affiliated with University of CAS, Nanjing and Nanjing Institute of Software Technology.

Authors' Contact Information: Haonan Yu, Institute of Software at Chinese Academy of Sciences, Beijing, China, yuhaonan23@otcaix.iscas.ac.cn; Jiaxin Zhu, Institute of Software at Chinese Academy of Sciences, Beijing, China, zhujiaxin@otcaix.iscas.ac.cn; Yingying Zheng, Institute of Software at Chinese Academy of Sciences, Beijing, China, zhengyingying14@otcaix.iscas.ac.cn; Yuwei Zhang, Institute of Software at Chinese Academy of Sciences, Beijing, China, zhangyuwei@otcaix.iscas.ac.cn; Wei Wang, Institute of Software at Chinese Academy of Sciences, Beijing, China, wangwei@otcaix.iscas.ac.cn; Jun Wei, Institute of Software at Chinese Academy of Sciences, Beijing, China, wj@otcaix.iscas.ac.cn; Tao Huang, Institute of Software at Chinese Academy of Sciences, Beijing, China, tao@otcaix.iscas.ac.cn.



ACM 2994-970X/2026/7-ARTFSE112
https://doi.org/10.1145/3808119

1 Introduction

The instruction set architecture (ISA) serves as the interface between hardware and software in a computer system [21]. It defines the low-level language through which software interacts with hardware. Over time, numerous ISAs have been developed, including widely adopted architectures such as x86 and ARM, as well as emerging alternatives like RISC-V. The ISA serves as a critical bridge between hardware and software in modern information systems. As such, ISA-related software development is a key component of the digital infrastructure of human society.

Each ISA has its own machine language and corresponding assembly language. High-level programming languages, such as C/C++, abstract away most of the architectural differences, enabling software to be developed in a largely platform-independent manner. Nevertheless, many low-level foundational software systems still include ISA-specific source code to handle ISA-dependent implementations. ISA-specific code provides ISA-tailored implementations that help maintain portability while achieving high performance on heterogeneous hardware. Figure 3 presents a simple example of ISA-specific code for the ARM and PowerPC ISAs. Moreover, many ISA-agnostic layers ultimately rely on ISA-specific backends directly or indirectly, making ISA-specific code indispensable in real-world systems. Notably, these files sometimes exhibit similar structures across different ISAs within the same project, as shown in Figure 3, suggesting the presence of similar code. This is partly because ISA-specific code is not isolated. It must interact with shared ISA-agnostic components and implement corresponding responsibilities under a common system design. Consequently, ISA-specific implementations across different ISAs often retain similarities in their structure and logic, even though they differ in irreducibly ISA-dependent details. This commonality may increase the complexity of software development and maintenance, introducing additional engineering overhead [32, 65].

Similar and identical code fragments are often referred to as code clones and have been widely studied in the literature [2, 13, 32, 39, 41, 45, 49, 65]. Although ISA-specific code is critical, it often constitutes a relatively small portion of the codebase and is maintained by a limited pool of experienced developers, which makes it less visible in day-to-day development. To the best of our knowledge, code similarity among ISA-specific code has not been systematically investigated to date. In prior studies, ISA-specific code is either absent or accounts for only a small fraction of the analyzed codebase. Little empirical evidence is available to guide researchers and practitioners engaged in multi-ISA software development. In this paper, we examine the ISA-specific code similarity, and explore the patterns of developing and maintaining the code. We collected 20 popular open-source software projects containing multiple-ISA-specific code for our study. From these projects, we extracted C/C++ files located in directories specifically designated for supporting multiple ISAs. These files are referred to as arch files. First, we identified the roles of arch files and to understand why they require multiple separate implementations. Because of the large number of arch files (18,900), we leverage three large language models to scale the identification of the roles and characteristics of these files. Next, we identified mirrored arch files whose file paths and names are similar across different ISAs. We then used GumTree [18], a widely adopted code differencing tool, to analyze similarities among mirrored file pairs and assess whether their separate implementations still exhibit commonalities. Finally, we examined the development and maintenance processes of the separate implementations to derive lessons on managing potential code commonalities across mirrored arch files.

We obtained three key findings: 1) The arch files mainly serve five roles closely tied to the ISA, such as ISA-specific hardware abstraction and ISA-specific code generation. 2) Although arch files are tightly coupled with the ISA, mirrored arch files often contain similar code. 3) Cross-ISA co-changes, where arch files for different ISAs are modified together within a single commit by

a developer working across these ISAs (as shown in Figure 4), occur frequently. Based on these findings, we discussed several implications for both practitioners and researchers to effectively manage arch files across different ISAs.

The main contributions of this paper are as follows:

(1) We present an empirical study of ISA-specific code similarities across multiple ISAs, filling a gap left by prior related studies.
(2) We offer empirical evidence to support researchers and practitioners in handling ISA-specific code similarities and advancing the development of corresponding tools.
(3) We provide a curated dataset of ISA-specific code variants, consisting of 6,464 mirrored arch files from 20 open-source foundational projects across eight ISAs, to support future research in this subarea.

In the remainder of this paper, we introduce essential background and review related work in Section 2, raise research questions in Section 3, outline our research approach in Section 4 to answer the questions, and present the results in Section 5. We discuss the implications for research and practice in Section 6. We present threats to the validity of our findings in Section 7 and conclude the paper in Section 8.

2 Background and Related Work

Our study involves two widely discussed topics in both practice and literature: instruction set architectures and code clones (similar or identical code).

2.1 Instruction Set Architecture

The Instruction Set Architecture (ISA) defines the interface between software and a processor, encompassing data types, the instruction set, registers, addressing modes, memory behavior, exception handling, and I/O interfaces.

Recent research has explored various aspects of ISAs, focusing on their design, compatibility, and impact on software systems. For instance, Patterson and Hennessy's foundational work on RISC architectures laid the groundwork for performance-oriented ISA design [21]. Subsequent studies have investigated cross-ISA binary translation [16, 19, 22, 61, 67], compiler optimizations tailored to specific ISAs [37], and the challenges of maintaining ISA-agnostic software in heterogeneous environments [6]. Moreover, the emergence of open ISAs like RISC-V has prompted renewed interest in customizable and extensible ISA design, enabling greater flexibility in both academic and industrial contexts [5, 14, 46, 60, 62]. These studies highlight the critical role ISAs play in shaping software development, deployment, and performance across platforms.

Today, most software projects are developed in high-level programming languages, which largely shield developers from ISA differences and allow their software to run on multiple ISAs. However, for lower-level software such as operating systems or compilers, separate code is often developed and maintained to support different ISAs. Despite its importance, knowledge about engineering ISA-specific code remains limited in the literature.

2.2 Code Clone and Code Similarity

2.2.1 Code Clone Levels. Code clone refers to a situation where two code fragments exhibit a certain degree of similarity under a given definition, indicating that some level of duplication exists between them. A variety of clone code definitions have already been proposed [8, 25, 49], Among them, the most widely accepted definition was proposed by [8]: If two code snippets conform to a certain similarity definition, then they are cloned code. Based on the degree and nature of similarity, syntactic code clones are commonly divided into four types[9, 12, 17, 28, 29, 31, 40, 49, 58]:

- **Type-1**: Identical code except for differences in whitespace and comments.
- **Type-2**: Syntactically identical code with variations in identifiers such as variable names, types, or function names.
- **Type-3**: Copied code that has been further modified, for example, through changes, additions, or deletions of statements.
- **Type-4**: Code fragments that implement the same functionality but differ significantly in both syntax and structure.

2.2.2 Code Clone Detection. Currently, detection of Type-1, Type-2 and Type-3 clones is relatively mature in academia, and a variety of tools have been developed to detect code clones.

Text-based methods treat source code as plain strings, employing techniques such as the longest common substring algorithm, local sequence alignment [54] or latent semantic analysis (LSA). Representative tools include NICAD [50] (which applies textual normalization for noise reduction and performs line-level comparison) and PlaGate [15] (an LSA-based code-plagiarism detector).

Token-based methods first convert code into token sequences and then search for common subsequences. JPlag [44] employs the Greedy String Tiling algorithm to identify the maximal set of contiguous token subsequences, focusing specifically on source-code plagiarism detection. SourcererCC [51] adopts an optimized inverted-index structure, together with token-ordering-based filtering heuristics, to reduce comparison overhead and enable scalable clone detection on large codebases.

AST-based methods construct and compare abstract syntax trees (ASTs) or their subtrees to detect clones. For example, DECKARD [23] uses tree-edit distance to measure structural similarity. GumTree [18] uses a two-phase algorithm to establish a mapping between two ASTs and applies a quadratic optimal algorithm to generate the shortest edit script to transform one AST into another, along with the node matching results between the two ASTs.

Graph-based methods rely on program-dependence graphs (PDGs) to capture both data and control dependencies, enabling the detection of semantic similarity including Type IV clones. CCSharp [59] constructs a PDG for each code fragment to improve generation efficiency and applies graph-matching techniques for clone detection.

Furthermore, there are also some methods such as **Metric-based methods** [24, 38], **Test-based methods** [33, 34, 55], **Image-based methods** [63], **Learning-based methods** [7, 20, 26, 32, 36, 64] and **Hybrid methods** [1, 11, 52, 53, 66] have achieved promising results.

Despite early successes in identifying simple clones, these methods continue to face challenges in detecting Type III/IV clones, supporting cross-language detection, scaling to large codebases, and addressing the scarcity of comprehensive evaluation datasets.

2.2.3 Code Similarity Analysis. Multiple factors contribute to the emergence of similar or identical code in a software system throughout its life cycle, including development strategy, accidental similarity, etc. [47, 49]. At the file level, Ossher et al. studied more than 13,000 open-source Java projects and found that over 10% of files were clones and more than 15% of projects contained at least one cloned file, highlighting the prevalence of whole-file reuse in practice [42].

As an early seminal study on clone genealogies, Kim et al. formalized clone evolution and showed that refactoring is not always the right response to clones. Many clones are short-lived, and many long-lived clones are not easily refactorable [27]. Subsequent studies further showed that cloned fragments may either evolve independently or be maintained through consistent change propagation [30, 57]. When coordinated updates are expected, inconsistent changes can pose maintenance risks [65].

Recent work has studied cross-language code similarity/clone detection [3, 35]. While this line of research is valuable, it is less directly related to our setting than classical within-language clone analysis, because we focus on ISA-specific implementations organized as architecture modules and on mirrored arch files across ISAs within the same project.

Prior studies on code similarity and clone-related maintenance have achieved strong results and yielded valuable insights for general-purpose clones, supported by high-quality resources such as BigCloneBench[56]. However, they rarely examine similarity specifically within ISA-specific implementations. Even small divergences in ISA-specific code can lead to inconsistent fixes and subtle cross-platform regressions. In this work, we provide a dedicated empirical characterization of code similarity in ISA-specific code, highlighting where commonality arises, how it manifests across ISAs, and what maintenance risks it introduces. Our findings complement existing clone research by centering on ISA-specific settings and by deriving practical implications for maintaining necessary consistency among ISA-dependent variants over time.

3 Research Questions

To systematically understand the characteristics of ISA-specific code, its cross-ISA similarity, and the resulting maintenance risks, we pose the following three research questions.

While abstraction and unification are commonly employed to simplify software development and maintenance, many foundational software projects still rely on ISA-specific code. The rationale and necessity for maintaining such code have not been systematically explored in the literature. We aim to explain why separate ISA-specific implementations are necessary across projects. Accordingly, we formulate our first research question as follows:

RQ1: What necessitates separate implementations of arch files across multiple ISAs?

Based on our preliminary observations, some arch files appear to be replicated across different ISAs under identical or similar filenames. Investigating the code similarities among these files is worthwhile, as understanding this commonality may help improve the efficiency of maintaining arch files across multiple ISAs. This leads to our second research question:

RQ2: How many arch files are mirrored across different ISAs, and to what extent are they similar?

Furthermore, it is important to investigate the development and maintenance patterns of mirrored arch files across ISAs, as this may reveal inefficiencies and provide insights for improvement. To deepen this investigation, we propose the third research question:

RQ3: How have the mirrored arch files been changed over time?

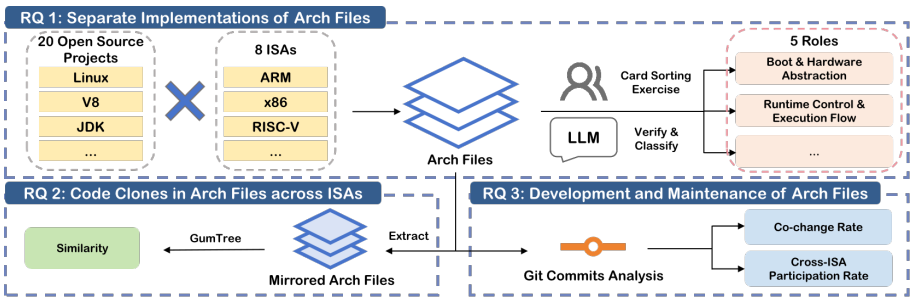


Fig. 1. Overview of the method used to address the research questions.

4 Methodology

The overview of the experimental process we conducted is described in Figure 1.

4.1 Studied ISAs and Projects

4.1.1 ISAs. Our study focuses on the most popular and actively maintained ISAs. We identified these ISAs by analyzing the Linux kernel, which serves as a representative fundamental software system that directly interfaces with hardware and is widely used across ISAs. The Linux Kernel project includes support for a broad set of widely used ISAs. Code specific to each ISA is primarily organized under the *arch/* directory of the Linux Kernel project, with individual subdirectories for each supported ISA. We analyzed the number of commits to these *arch/** directories in the past three years (2022-2024) and identified the most actively maintained ISAs.

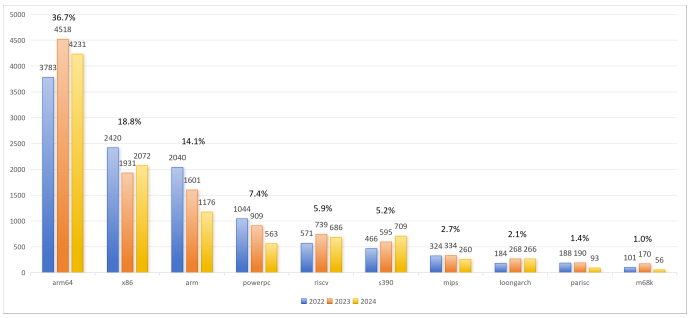


Fig. 2. The number of commits to the *arch/** directories of Linux Kernel project.

The number and proportion of commits involving each ISA are shown in Figure 2. Note that there are 20 ISAs supported by Linux Kernel now, to save space, we only show the results of the top 10 ISAs. We observe that the top eight ISAs account for more than 90% of the commits to the *arch* directory. Furthermore, the eighth ISA has more than 1.5 times as many commits as the ninth. Consequently, we selected the eight most popular and actively maintained ISAs and presented them in descending order based on commit count, as follows: *ARM64*, *ARM*, *x86*, *PowerPC*, *RISC-V*, *S390*, *MIPS*, and *LoongArch*.

The names of ISAs are sometimes written in different forms, such as *ARM64* and *AArch64*. In this paper, we follow the naming convention used by the Linux Kernel project to maintain consistency.

4.1.2 Projects. To conduct our study, we selected 20 representative open-source projects that include ISA-specific code for at least four of the eight architectures identified in Section 4.1.1. Table 1 presents a summary of the collected projects. The architectures supported by the projects are listed in the last column.

We selected the versions from the default branches of all projects as of the data collection date, April 8, 2025. The Git repository URLs and the short commit IDs corresponding to these versions are listed in Table 1.

All the ISA-specific code in these projects is primarily implemented in C/C++. They have gained significant popularity, each amassing over 1,000 stars on GitHub, and span five distinct technical domains, as detailed below.

- Operating systems and firmware: Linux, Zephyr, U-Boot, coreboot.
- Compilers and language runtimes: GCC, LLVM, V8, OpenJDK, LuaJIT, Mono.
- Core libraries and system interfaces: glibc, libffi, libunwind, cpuinfo.
- Binary analysis and emulation: QEMU, Unicorn, radare2, Capstone, Keystone.
- Multimedia processing: FFmpeg.

Table 1. Overview of the 20 Studied Software Projects

Project	Description	Code Lines	Commit ID	Stars	Archs[1]
Linux	Linux is an open-source, Unix-like operating system kernel and ecosystem. https://github.com/torvalds/linux *[Operating systems and firmware]*	27.1M	0af2f6be1b42	191k	1-8
FFMpeg	FFmpeg is a collection of libraries and tools to process multimedia content such as audio, video, subtitles and related metadata. https://github.com/FFmpeg/FFmpeg *[FFmpeg]*	1.5M	02eda84bf2	49k	1-6,8
LLVM	A toolkit for the construction of highly optimized compilers, optimizers, and run-time environments. https://github.com/llvm/llvm-project *[Compilers and language runtimes]*	8.4M	03f21e2ba3be	31.7k	1-8
V8	V8 is Google's open source JavaScript engine. https://github.com/v8/v8 *[Compilers and language runtimes]*	1.3M	ab2f77b6b1e	24k	1-8
radare2	r2 is a featureful low-level command-line tool that can edit files on local hard drives, view kernel memory, and debug programs locally or via a remote gdb/windbg server. https://github.com/radareorg/radare2 *[Binary analysis and emulation]*	807k	245cb7204d	21.4k	1-8
JDK	The Java Development Kit (JDK) is a comprehensive software development environment — including the Java Runtime Environment (JRE), compilers, and core libraries. https://github.com/openjdk/jdk *[Compilers and language runtimes]*	4.2M	9844c1c52b9	20.7k	1-5,7
Zephyr	The Zephyr Project is a scalable real-time operating system (RTOS) supporting multiple hardware architectures, optimized for resource-constrained devices, and built with security in mind. https://github.com/zephyrproject-rtos/zephyr *[Operating systems and firmware]*	2.3M	ddedff71557	11.9k	1-3,5,6
Mono	Mono is a software platform designed to allow developers to easily create cross-platform applications. It is an open-source implementation of Microsoft's .NET Framework. https://github.com/mono/mono *[Compilers and language runtimes]*	560k	0f53e9e151d	11.2k	1-8
QEMU	QEMU is a generic and open source machine & userspace emulator and virtualizer. https://github.com/qemu/qemu *[Binary analysis and emulation]*	1.8M	dfaecc04c4	11.1k	1-8
GCC	GCC is the GNU Compiler Collection, used to compile source code written in various programming languages into executable programs. https://github.com/gcc-mirror/gcc *[Compilers and language runtimes]*	10.2M	a9bbb60b7ca	9.8k	1-8
Unicorn	Unicorn is a lightweight, multi-platform, multi-architecture CPU emulator framework, based on QEMU. https://github.com/unicorn-engine/unicorn *[Binary analysis and emulation]*	431.8k	64c72267	8k	1-7
Capstone	Capstone is a disassembly framework with the target of becoming the ultimate disasm engine for binary analysis and reversing in the security community. https://github.com/capstone-engine/capstone *[Binary analysis and emulation]*	349.8k	72f7d305	7.9k	1-8
LuaJIT	LuaJIT is a Just-In-Time (JIT) compiler for the Lua programming language. https://github.com/LuaJIT/LuaJIT *[Compilers and language runtimes]*	77.1k	e0a7ea8a	5k	1-4,6
U-Boot	U-Boot, a bootloader for embedded boards based on PowerPC, ARM, MIPS and several other processors. https://github.com/u-boot/u-boot *[Operating systems and firmware]*	2.9M	34820924edb	4.4k	1-6
libffi	The libffi library provides a portable, high-level programming interface to various calling conventions. https://github.com/libffi/libffi *[Core libraries and system interfaces]*	37.2k	bb1a84e	3.3k	1-8
Keystone	Keystone is a lightweight multi-platform, multi-architecture assembler framework. https://github.com/keystone-engine/keystone *[Binary analysis and emulation]*	118.6k	fb92f32	2.4k	1-7
coreboot	coreboot is a Free Software project aimed at replacing the proprietary firmware (BIOS/UEFI) found in most computers. https://github.com/coreboot/coreboot *[Operating systems and firmware]*	1.5M	0b7b68389a	2.3k	1-5,8
glibc	The GNU C Library is the standard system C library for all GNU systems, and is an important part of what makes up a GNU system. https://github.com/bminor/glibc *[Core libraries and system interfaces]*	1.3M	fb3d821afa	1.6k	1-8
cpuinfo	cpuinfo is a library for detecting information about the host CPU that is essential for performance optimization. https://github.com/pytorch/cpuinfo *[Core libraries and system interfaces]*	17.2k	39ea79a	1.1k	1,2,3,5
libunwind	libunwind is a portable and efficient C API for determining the current call chain of ELF program threads of execution and for resuming execution at any point in that call chain. https://github.com/libunwind/libunwind *[Core libraries and system interfaces]*	32.0k	405b13b3	1.1k	1-8

[1] Table 2 lists the architectures corresponding to the IDs.

4.2 Arch Files

File structures in software systems typically reflect the functionalities of the files. In the studied projects, ISA-specific code is generally organized into dedicated files located in specific directories. These files or directories are named after their respective ISAs and arranged in parallel. We refer to these files as ***arch files***.

ISA-related code can be organized in other ways, such as being scattered throughout the codebase. In this paper, however, we focus only on the explicit pattern of arch files. In addition, we consider only the main code of the projects, excluding tests, third-party libraries, and other auxiliary

components. According to the knowledge base we built during our ISA-related development[1], the terms we derived for the studied ISAs (case-insensitive) are shown in Table 2.

Table 2. CPU Architectures and Their Aliases

	Architecture	Aliases
1	ARM	*arm, aarch, arm32, aarch32*
2	ARM64	*arm64, aarch64*
3	x86	*x86, x64, x86_64, x86-64, ia64*
4	PowerPC	*ppc, powerpc, ppc32, powerpc32, ppc64, powerpc64*
5	RISC-V	*riscv, risc-v, riscv32, risc-v32, riscv64, risc-v64*
6	MIPS	*mips, mips32, mips64*
7	S390	*s390, systemz*
8	LoongArch	*loong, loongarch,loong64,loongarch64*

Using these keywords, we extracted a total of 18,900 C/C++ source code files named after the ISA.

4.3 Arch File Roles Identification

To understand why separate implementations of arch files are necessary, we analyze and identify the roles played by these files. Following a widely used manual analysis approach [43], three authors performed a pilot analysis on a representative sample to examine the roles of arch files. They have at least three years of expertise in x86, ARM, ARM64, MIPS, and RISC-V. The annotation does not require deep knowledge of every individual ISA. A general architectural understanding across ISAs is sufficient, for example, recognizing common architectural components such as register organizations and calling conventions. The annotations were performed independently by each annotator, and we subsequently conducted an inter-rater reliability check among the three annotators. Disagreements were resolved through discussion and voting (with minority views yielding to the majority). In most cases, these discussions led to a clear consensus.

The sample size of 385 files was determined to achieve a 95% confidence level with a 5% margin of error, based on a standard sample size calculator. The authors reviewed the samples and, through discussion, defined the roles and their descriptions. The completeness of these manually identified roles was further evaluated with the assistance of large language models (LLMs).

We employed three cost-effective LLMs, i.e., GPT-4.1-mini (https://api.ifopen.ai/v1, *gpt-4.1-mini*) Llama-4-scout (https://api.ifopen.ai/v1, *llama-4-scout*) and DeepSeek-R1 (https://uni-api.cstcloud.cn/v1, *deepseek-r1:671b-0528*) to classify the arch files by their roles. We accessed these models via cloud-hosted LLM APIs. The corresponding providers and model identifiers are given in parentheses after each model name. In light of the lessons from Renze and Guven[48], we set the temperature parameter to 0 and left all other adjustable inference parameters at their default values to ensure deterministic outputs. We formed the prompt as follows:

> You are an expert software engineer analyzing the Instruction-Set-Architectures-specific files. You are given a code file along with its file path and main content. Your task is to classify the file's role into one of the predefined main categories or the miscellaneous category.
> File path: [file path]
> File content: [file content]
> Candidate Roles: [role 1, role 2, ...]

If the content of a file exceeds the permitted length, it is truncated, but the function signatures from the truncated portion are still included.

[1]https://wiki.rvpt.top/

We aggregated the classification results from the three LLMs. When all three models agreed on a file's category, we directly assigned the document to that category. In cases with a 2:1 split, we applied a majority-voting strategy, assigning the final label to the category chosen by the majority. When no agreement was reached (each LLM produced a different classification), we manually determined the final category.

We further conducted two checks to verify the reliability of this automated strategy. First, we compared our results with those of the LLMs for the 385 arch files used in the pilot analysis. Second, we manually examined a sample of 385 cases with a 2:1 split.

4.4 Arch File Similarity Analysis

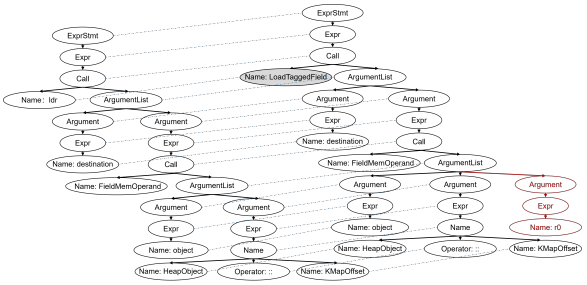
To analyze arch file similarities, we first determined the pairs of arch files to compare and subsequently measured their similarity.

4.4.1 Compared Pairs. To enable consistent and scalable similarity analysis, we focus on ***mirrored arch files***, defined as the pairs of files whose file paths differ solely in the ISA name. For example, the files *v8/src/deoptimizer/arm64/deoptimizer-arm64.cc* and *v8/src/deoptimizer/riscv/deoptimizer-riscv.cc*, together with *llvm-project/llvm/include/llvm/TargetParser/x86TargetParser.h* and *llvm-project/llvm/include/llvm/TargetParser/PPCTargetParser.h*, constitute two pairs of mirrored arch files. The mirrored arch files are likely to share core logic in their design, differing only in ISA-specific details (register usage, calling conventions, etc.), and thus exhibit the highest degree of semantic commonality. In some cases, a file in a mirrored pair is initially created by copying its counterpart. Our similarity comparison focuses exclusively on the mirrored arch files. In contrast to arbitrary arch file pairs, similar code fragments found in mirrored arch files are more indicative of intentional code clones. This also delivers acceptable performance on the scale of 20 projects and 8 ISAs.

We introduced a path-scanning approach for identifying mirrored arch files. It should be noted that, in C/C++ programs, function prototypes, macros, constants, global variables, and data structures can be defined either in header files or directly in source files. To align mirrored arch files across ISAs, we merged header and source files if any counterpart places header-like content directly in a source file.

Additionally, in order to assess the proportion of code volume occupied by the arch files and mirrored arch files in each open-source project, we employed the Succinct Code Counter (SCC)[10] tool to perform an exact count of the effective lines of code (LOC) across all source code files in the projects.

4.4.2 Similarity Calculation. We found that code in arch files has several characteristics that may interfere with code similarity calculation. First, identifiers, such as function or class names, frequently incorporate the ISA name or are derived from registers or instructions. Second, implementations of the same functionality may vary in both the number and types of arguments. Third, similar code fragments may exhibit variations due to factors such as differences in data structures, the use of intrinsic functions, and memory models. These variations severely hinder basic text- or line-level similarity metrics, necessitating a method that can effectively handle them to achieve accurate results. AST-based similarity calculation, which ignores textual differences, focuses solely on node types and structural relationships, better captures semantic homology. AST-based methods have been shown to accurately recognize type I, II, and III code clones [65], and are capable of handling variations across ISAs. Accordingly, we compute arch file similarity using GumTree [18], one of the most widely adopted AST differencing tools (over 1,100 GitHub stars and actively maintained) [4]. It provides fine-grained matching that is well aligned with our goal of quantifying structural similarity between mirrored arch files, and it yields interpretable mappings that we can manually inspect for sanity checks. The version of GumTree used in this study builds upon the recent work




```
#v8/src/codegen/arm/macro-assembler-arm.cc
void MacroAssembler::LoadMap(
  Register destination, Register object){
    ldr(destination, FieldMemOperand(
      object, HeapObject::kMapOffset));
    }

#v8/src/codegen/ppc/macro-assembler-ppc.cc
void MacroAssembler::LoadMap(
  Register destination, Register object){
    LoadTaggedField(destination, FieldMemOperand(
      object, HeapObject::kMapOffset),r0);
    }
```

Fig. 3. Code fragments extracted from mirrored arch files and their AST differences.

of Falleri and Martinez [18], which introduces an optimal algorithm and has been evaluated as state-of-the-art. This enables precise computation of arch file similarity. Note that GumTree, as a tree-matching algorithm, does not directly output a similarity score, and we perform the following steps:

(1) **Preprocessing & Concatenation:** For each mirrored group's header and source files, concatenate them in a fixed order to form a single composite file for comparison.
(2) **AST Parsing & Node Counting:** Use GumTree's *parse* function to convert each file into an AST, recording its total node count $|A|$ and $|B|$.
(3) **Diff & Match Extraction:** Run GumTree's *diff* function on any two files in the same parallel group to obtain the number of matched nodes $|A \cap B|$.
(4) **Similarity Calculation:** Calculate the Jaccard similarity:

$$\mathrm{Sim}(A, B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

to quantify the semantic homology of each pair of mirrored files.

Figure 3 illustrates an example of two code fragments extracted from mirrored arch files for similarity computation. The code corresponding to the AST nodes is highlighted in bold. The left AST corresponds to the upper snippet, and the right AST to the lower snippet. The structural differences between the two ASTs have been explicitly annotated. Specifically, the right AST differs from the left in two respects: variable names and the introduction of an additional parameter. For similarity measurement, the only effective difference is the argument r0 in the function call. The similarity between the two highlighted code fragments is computed according to the following formula: $Sim = \frac{22}{22+25-22} = 88\%$

While computing the average similarity at the project level or over all 20 projects, we used the product of the number of AST nodes in the two files as their weights, and performed a weighted average over all mirrored file pairs. The formula is as follows:

$$\mathrm{AvgSim} = \frac{\sum s_{ij} \cdot (n_i \cdot n_j)}{\sum n_i \cdot n_j}$$

where s_{ij} is the similarity between file i and j, n_i is the number of AST nodes in file i.

4.5 Arch File Changes Analysis

We investigated the development and maintenance of the mirrored arch files by analyzing the commit history of the Git repositories. The study focuses on a recent three-year period, from 2022 to 2024. During this period, most of the projects maintained active support for the studied ISAs, particularly emerging ISAs such as RISC-V. It is worth noting that Keystone has been inactive since

```
--- a/src/baseline/arm64/baseline-assembler-arm64-inl.h
+++ b/src/baseline/arm64/baseline-assembler-arm64-inl.h
@@ -557,7 +557,7 @@ void BaselineAssembler::EmitReturn(MacroAssembler* masm) {
     __ LoadContext(kContextRegister);
     __ LoadFunction(kJSFunctionRegister);
     __ masm()->PushArgument(kJSFunctionRegister);
-    __ CallRuntime(Runtime::kBytecodeBudgetInterruptFromBytecode, 1);
+    __ CallRuntime(Runtime::kBytecodeBudgetInterrupt, 1);

     __ masm()->Pop(kInterpreterAccumulatorRegister, params_size);
     __ masm()->SmiUntag(params_size);
```

```
--- a/src/baseline/loong64/baseline-assembler-loong64-inl.h
+++ b/src/baseline/loong64/baseline-assembler-loong64-inl.h
@@ -449,7 +449,7 @@ void BaselineAssembler::EmitReturn(MacroAssembler* masm) {
     __ LoadContext(kContextRegister);
     __ LoadFunction(kJSFunctionRegister);
     __ masm()->Push(kJSFunctionRegister);
-    __ CallRuntime(Runtime::kBytecodeBudgetInterruptFromBytecode, 1);
+    __ CallRuntime(Runtime::kBytecodeBudgetInterrupt, 1);

     __ masm()->Pop(params_size, kInterpreterAccumulatorRegister);
     __ masm()->SmiUntag(params_size);
```

```
--- a/src/baseline/mips64/baseline-assembler-mips64-inl.h
+++ b/src/baseline/mips64/baseline-assembler-mips64-inl.h
@@ -459,7 +459,7 @@ void BaselineAssembler::EmitReturn(MacroAssembler* masm) {
     __ LoadContext(kContextRegister);
     __ LoadFunction(kJSFunctionRegister);
     __ masm()->Push(kJSFunctionRegister);
-    __ CallRuntime(Runtime::kBytecodeBudgetInterruptFromBytecode, 1);
+    __ CallRuntime(Runtime::kBytecodeBudgetInterrupt, 1);

     __ masm()->Pop(params_size, kInterpreterAccumulatorRegister);
     __ masm()->SmiUntag(params_size);
```

```
--- a/src/baseline/riscv64/baseline-assembler-riscv64-inl.h
+++ b/src/baseline/riscv64/baseline-assembler-riscv64-inl.h
@@ -478,7 +478,7 @@ void BaselineAssembler::EmitReturn(MacroAssembler* masm) {
     __ LoadContext(kContextRegister);
     __ LoadFunction(kJSFunctionRegister);
     __ masm()->Push(kJSFunctionRegister);
-    __ CallRuntime(Runtime::kBytecodeBudgetInterruptFromBytecode, 1);
+    __ CallRuntime(Runtime::kBytecodeBudgetInterrupt, 1);

     __ masm()->Pop(params_size, kInterpreterAccumulatorRegister);
     __ masm()->SmiUntag(params_size);
```

```
--- a/src/baseline/s390/baseline-assembler-s390-inl.h
+++ b/src/baseline/s390/baseline-assembler-s390-inl.h
@@ -568,7 +568,7 @@ void BaselineAssembler::EmitReturn(MacroAssembler* masm) {
     __ LoadContext(kContextRegister);
     __ LoadFunction(kJSFunctionRegister);
     __ Push(kJSFunctionRegister);
-    __ CallRuntime(Runtime::kBytecodeBudgetInterruptFromBytecode, 1);
+    __ CallRuntime(Runtime::kBytecodeBudgetInterrupt, 1);

     __ Pop(kInterpreterAccumulatorRegister, params_size);
     __ masm()->SmiUntag(params_size);
```

```
--- a/src/baseline/x64/baseline-assembler-x64-inl.h
+++ b/src/baseline/x64/baseline-assembler-x64-inl.h
@@ -440,7 +440,7 @@ void BaselineAssembler::EmitReturn(MacroAssembler* masm) {
     __ LoadContext(kContextRegister);
     __ Push(MemOperand(rbp, InterpreterFrameConstants::kFunctionOffset));
-    __ CallRuntime(Runtime::kBytecodeBudgetInterruptFromBytecode, 1);
+    __ CallRuntime(Runtime::kBytecodeBudgetInterrupt, 1);

     __ Pop(kInterpreterAccumulatorRegister, params_size);
     __ masm()->SmiUntag(params_size);
```

Fig. 4. An illustrative example of cross-ISA co-change in V8.

its last update in 2023, and similarly, Mono ceased to be maintained as an independent project in August 2024. Therefore, we excluded both projects from this historical analysis. We collected the commits made to the mirrored arch files in this period, and analyzed these commits from two aspects:

(1) **Co-changes across ISAs.**
Figure 4 illustrates a cross-ISA co-change example from the V8 project (commit ID: `8039e2715ad82c678952c6fbbcf39b10c1985d3a`), where mirrored arch files for ARM64, LoongArch, MIPS, RISC-V, S390, and x86 are modified within this single commit. The figure shows that these arch files remain highly similar both before and after the commit, and that the parameters of the *__CallRuntime* macro are updated consistently across all of them.
This change-synchronization dimension captures whether mirrored files for different ISAs are updated together in the same commits, reflecting the degree of coordinated maintenance and the timely propagation of functionally equivalent updates.
We extracted commits that modified files across multiple ISAs and examined the frequency of such co-changes between each pair of ISAs, i.e., co-change rate. In particular, the co-change rate between each pair of ISAs is calculated as follows. For a given pair of ISAs, *ia* and *ib*, we collected the set of commits that modify mirrored files in each ISA, resulting in two sets, C_{ia} and C_{ib}. The intersection of these sets represents the commits that simultaneously modify mirrored files in both ISAs. The co-change rate is then calculated as the Jaccard coefficient of these two commit sets:

$$J(C_{ia}, C_{ib}) = \frac{|C_{ia} \cap C_{ib}|}{|C_{ia} \cup C_{ib}|}$$

We use a weighted mean to report the project-level co-change rate. The weights are based on the product of the cardinals of the two commit sets. Therefore, the calculation formula of the project-level co-change rate is:

$$\bar{J}_{commit} = \frac{\sum_{1 \le i < j \le 8} |C_i|\,|C_j|\,J(C_i, C_j)}{\sum_{1 \le i < j \le 8} |C_i|\,|C_j|}$$

(2) **Cross-ISA participation.**

Cross-ISA participation captures whether the same developers work across ISA-specific mirrored file sets, reflecting cross-ISA knowledge sharing and the project's organizational structure.

We treated the mirrored files within an ISA as a file set. Developers who have committed to an ISA's file set are considered members of its team. In particular, the Cross-ISA participation between each pair of ISAs was calculated as follows. For a given pair of ISAs, *ia* and *ib*, we add the developer to the set D_{i_a} (or D_{i_b}) if they once modified mirrored files in each ISA. The intersection of these sets indicates the number of developers involved across the ISAs. The cross-ISA participation rate is then calculated as the Jaccard coefficient of these two developer sets:

$$J(D_{ia}, D_{ib}) = \frac{|D_{ia} \cap D_{ib}|}{|D_{ia} \cup D_{ib}|}$$

We use a weighted mean to report the project-level cross-ISA participation rate. The weights are based on the product of the cardinals of the two developer sets. Therefore, the calculation formula of the project-level cross-ISA participation rate is:

$$\bar{J}_{developer} = \frac{\sum_{1 \le i < j \le 8} |D_i|\,|D_j|\,J(D_i, D_j)}{\sum_{1 \le i < j \le 8} |D_i|\,|D_j|}$$

5 Results

5.1 Roles of Arch Files

From the sample set, we manually identified five roles performed by the arch files. We introduce the roles as follows.

(1) Boot & Hardware Initialization: Start and configure hardware at boot, including CPU/SoC probing, buses, devices, memory maps, and hardware definition information.
(2) Runtime Control & Execution Flow: Manage program execution, including exceptions, system calls, call conventions, stack frames, and context switching.
(3) Memory Management & Virtualization. Handle memory subsystems, page tables, caching, and virtualization support.
(4) Instruction Processing & Code Generation. Map and transform instructions, manage operands, register allocation, machine instructions decoding, and scheduling.
(5) Core Algorithms & Calculations. Implement reusable algorithms, math libraries, multimedia processing, cryptography, compression, and data encoding/decoding.

We evaluate the completeness of the identified roles using the method described in Section 4.3. Based on our role taxonomy, the three LLMs were able to reach a consensus on 13,006 (68.8%) arch files. For 5,150 (27.2%) arch files, the outcomes of the LLMs exhibited a 2:1 split. The LLMs yielded three different outcomes for the remaining 744 (3.9%) arch files. Using the strategy outlined in Section 4.3, we determined the roles of all 18,900 files, which are summarized in Table 3. Furthermore, we confirmed that the final results for the 385 arch files in the pilot analysis, as well as for cases with a two-to-one agreement among the three LLMs, matched the manually obtained results. In addition, no inconsistencies were observed when the LLMs were configured with a temperature of 0.

Overall, our role taxonomy proved highly complete: in the 20 projects, more than 90% of the arch files were placed into a main category, and in 19 projects the rate exceeded 95%. Even in the lowest-scoring case (QEMU), the classification success rate was still above 92%, demonstrating that the identified roles capture the vast majority of ISA-related code.

In these 20 projects, arch files categorized as miscellaneous constitute approximately 2.1% of the total. Further observation indicates that these arch files primarily consist of placeholder code

Table 3. Roles of Arch Files in the 20 Studied Projects

Project	Boot & Hardware Initialization	Runtime Control & Execution Flow	Memory Management & Virtualization	Instruction Processing & Code Generation	Core Algorithms & Calculations	Miscellaneous	Total	Total LOC
Linux	3,994	1,498	1,167	276	611	191	7,737	1,596,905
FFmpeg	18	0	0	5	550	2	575	155,645
LLVM	4	55	0	786	9	0	854	533,532
V8	0	58	0	257	5	0	320	340,898
radare2	3	68	0	80	4	2	157	101,615
JDK	36	232	81	386	30	19	784	260,149
Zephyr	125	145	52	5	13	7	347	27,745
Mono	8	23	0	35	0	0	66	57,819
QEMU	545	229	112	144	27	81	1,138	407,417
GCC	13	47	1	273	63	6	403	254,136
Unicorn	56	38	24	106	10	1	235	233,462
Capstone	0	0	0	118	0	1	119	69,650
LuaJIT	0	0	0	15	0	0	15	14,152
U-Boot	2,712	66	77	18	40	15	2,928	522,600
libffi	0	1	0	28	0	0	29	9,338
Keystone	0	5	0	132	0	0	137	41,042
coreboot	223	52	27	17	11	3	333	31,691
glibc	96	499	34	29	1,598	76	2,332	87,032
cpuinfo	45	0	1	0	1	0	47	14,985
libunwind	0	341	0	2	1	0	344	13,500
Total	**7,878**	**3,357**	**1,576**	**2,712**	**2,973**	**404**	**18,900**	**4,773,313**

used during development, deprecated legacy code, references to other files without functional implementation, and auxiliary code for developer debugging output.

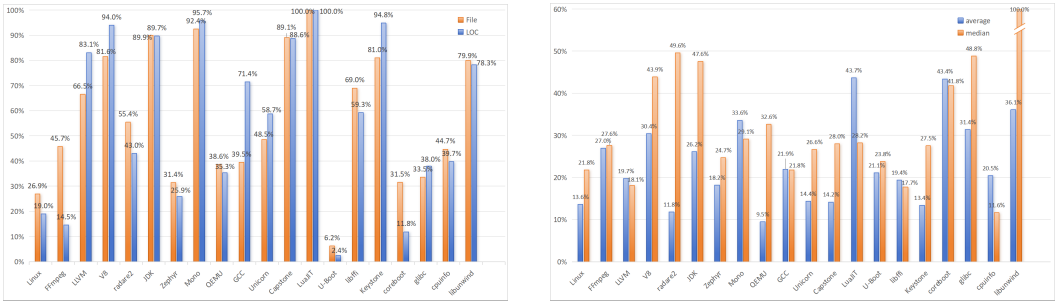
Since the technical scope of each studied project differs, the relative proportions of the five roles vary across projects, as shown in Table 3.

To illustrate the roles of arch files, we highlight four representative projects.

- **Linux** As a general-purpose operating system supporting numerous platforms, Linux shows a relatively balanced distribution of arch functionality. More than 50% of its correctly classified arch files fall into Role 1: Boot & Hardware Initialization, reflecting the kernel's need to probe, configure, and initialize diverse hardware.
- **glibc** The GNU C Library provides the standard C runtime, including optimized implementations of mathematical routines. Consequently, files of Role 5: Core Algorithms & Calculations are particularly significant in glibc, comprising more than 60% of its arch files.
- **JDK** This codebase encompasses the Java Runtime Environment, compilers, and core class libraries. Accordingly, large portions of its arch files implement Role 2: Runtime Control & Execution Flow and Role 4: Instruction Processing & Code Generation, since JVM bytecode translation and execution heavily depend on ISA-specific scheduling and stack management logic.
- **U-Boot** As a bootloader, U-Boot is primarily responsible for hardware bring-up. More than 90% of its arch files are classified as Role 1.

These roles are clearly tied to specific ISAs, which highlight the necessity of developing and maintaining separate ISA-specific code when a software project supports multiple ISAs.

Answer to RQ1: Separate implementations of arch files are necessary because ISA-specific code repeatedly appears in several recurring roles, including hardware initialization, runtime control, memory management, instruction processing, and core computations. These parts directly depend on ISA-level constraints such as instruction semantics, calling conventions, register organization, and memory mechanisms.



(a) The proportion of mirrored arch files relative to all arch files.

(b) The average and median similarity of mirrored arch files.

Fig. 5. Mirrored arch files and their similarity across projects.

5.2 Arch File Similarity across ISAs

5.2.1 Mirrored Arch files. Using the method described in Section 4.4.1, we quantify the prevalence of mirrored patterns within the arch files of the 20 studied projects. We identified a total of 6,464 mirrored arch files. The proportion of mirrored arch files relative to all arch files is summarized in Figure 5a.

The overall arithmetic mean of mirrored arch file proportion is 57.6%, and when weighted by file counts within each project, it drops to 34.2%. The overall arithmetic mean of lines of code (LOC) proportion in arch file is 57.2%, which is almost consistent with the file level proportion.

From a project perspective, the proportion ranges from a mere 6.2% in U-Boot, where arch files are largely limited to hardware bring-up code, to 100.0% in LuaJIT, whose ISA modules are completely mirrored.

Notably, operating systems and firmware projects such as Linux, coreboot, Zephyr, and especially U-Boot cluster at the lower end of the spectrum. This reflects the fact that their arch files are dominated by boilerplate code and chip-initialization routines, which inherently provide limited opportunities for mirroring at the ISA level.

In contrast, compilers and language runtimes, including LuaJIT, V8, JDK and Mono, show very high ratios of mirrored code. More than 80% of arch files are mirrored, and all these projects have devoted a lot of effort to Role 4 (Instruction Processing & Code Generation).

These results suggest two practical takeaways. First, teams maintaining OS and firmware stacks can achieve broad ISA support with relatively minimal overhead from duplication. Second, compiler and JIT developers should anticipate the need for substantial expertise in handling mirrored patterns.

5.2.2 Similarity of Mirror Arch Files. Using the method described in Section 4.4.2, we calculated the pairwise similarity of the mirrored arch files.

The results in Figure 5b shows that similar code is widespread among mirrored arch files, and that weighted average similarities vary significantly across projects, ranging from **9.5%** (QEMU) to **43.7%** (LuaJIT):

The weighted average similarity across mirrored files in all 20 projects is 21.7%, but this rate varies significantly at the project-level. LuaJIT has relatively few arch files compared to other projects, yet it displays extremely high mirroring (100.0%) and similarity (43.7%), indicating that its small set of source files is highly uniform. Libunwind and glibc, both core libraries and system interfaces, rank third and fifth at 36.1% and 31.4%, respectively. Although glibc's mirrored code constitutes a smaller fraction of its arch files, this still indicates extensive code cloning within that

subset. Conversely, QEMU (9.5%) and radare2 (11.8%) exhibit the lowest similarities, suggesting that their ISA backends tend toward independent implementations rather than reusing cloned code.

In libunwind, the median similarity is 100.0%, because it contains many very short arch files whose ASTs are nearly identical, causing its median to stand out above all other projects. Radare2 (49.6%) and glibc (48.8%) follow closely, showing that at least half of their file pairs share nearly 50% similarity. cpuinfo (11.6%) and libffi (17.7%) display relatively low median similarities, but their average similarities are approximately equal to or even exceed their medians; LuaJIT (28.2%), LLVM (18.1%) and Mono (29.1%) exhibit the same pattern. This indicates that, in most projects, the larger (higher-weight) arch files tend to have lower similarity, while the smaller files exhibit higher similarity.

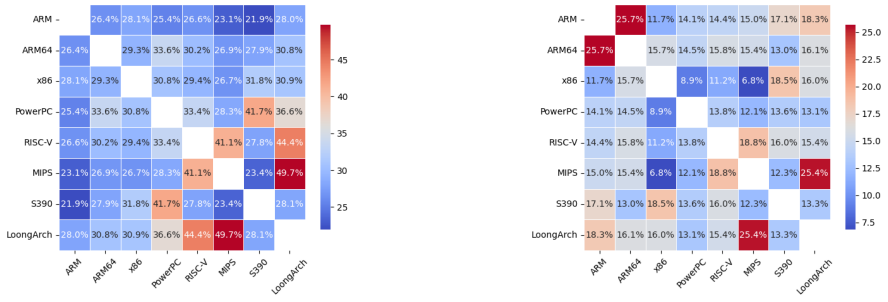


Fig. 6. The similarities between ISAs in V8 (left) and Linux (right).

We also computed the pairwise similarity between the mirrored arch files corresponding to each project's supported ISAs. Here, we have selected two representative cases for detailed analysis: V8, which exhibits relatively high overall similarity, and Linux, which shows comparatively low similarity. The data is described in Figure 6.

In the V8 project, the differences in code similarity across ISAs are quite pronounced. Notably, the MIPS–LoongArch–RISC-V trio displays relatively strong similarity scores (49.7%, 44.4% and 41.1%). This likely stems from the fact that both the RISC-V and LoongArch implementations were developed by modifying the existing MIPS codebase. At the same time, the PowerPC–S390 pair also shows a high similarity (41.7%) compared to other architectures. This may be because a single developer has long maintained both S390 and PowerPC simultaneously.

By contrast, in the Linux project, the absolute similarity values among various architectures do not differ significantly. The most notable pairings are ARM with ARM64 (25.7%) and LoongArch with MIPS (25.4%). These are probably explained by intrinsic ISA-level affinities: LoongArch's early design was based on MIPS, and later it evolved into its own instruction set while retaining compatibility with MIPS.

Answer to RQ2: Necessary ISA-specific separation coexists with substantial cross-ISA commonality. Across the 20 projects, 34.2% of arch files have mirrored counterparts, with a weighted average similarity of 21.7%. Notably, the average similarity reaches 43.7% in LuaJIT and 43.4% in coreboot. Some ISAs tend to exhibit higher similarity, such as MIPS, LoongArch, and RISC-V.

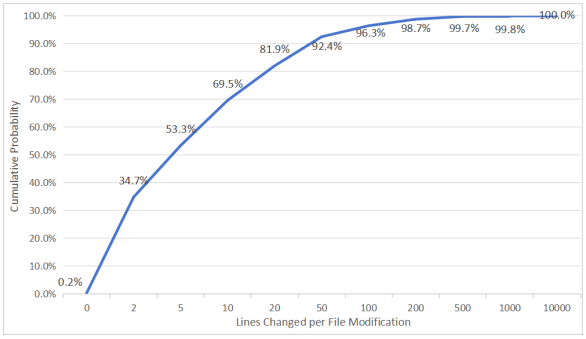


Fig. 7. Cumulative distribution of lines changed per file modification.

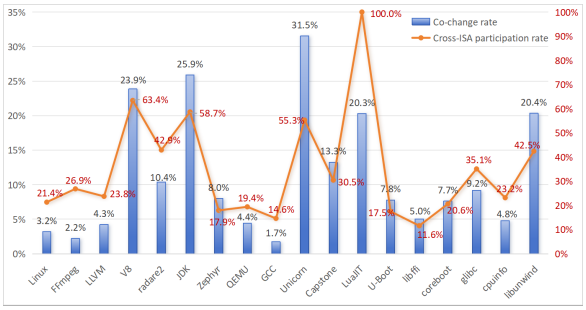


Fig. 8. The co-change rate and cross-ISA participation rate of the mirrored arch files.

5.3 Development and Maintenance of Arch Files

Using the method described in Section 4.5, we extracted 17,531 file changes among 2,778 co-changes of mirrored arch files. Each file change corresponds to modifications made to a file within a single commit. We characterized the co-changes by the number of non-formatting lines, where certain code tokens are added and/or deleted. The distribution of modified lines is shown in Figure 7. Among the co-changes, 65.3% of the file changes involve more than two lines, indicating that most changes are substantive rather than trivial. File changes consisting solely of formatting modifications were excluded from the subsequent analysis.

The results on co-changes and cross-ISA participation of the mirrored arch files are presented in Figure 8. Overall, all the projects demonstrate co-changes and cross-ISA participation in mirrored arch files between different ISAs. The weighted average co-change rate of the 20 projects was 7.2%, and the cross-ISA participation rate was 26.5%.

Projects Unicorn and JDK demonstrate a particularly high co-change rate of 31.5% and 25.9%, respectively. This suggests a high level of coupling between ISAs, where developers frequently modify multiple architectures in a single commit. Conversely, most projects show a relatively low co-change rate between ISAs. For example, GCC has a co-change rate of only 1.7%, indicating a tendency among developers to focus on modifying code specific to a single ISA at a time. This can cause inconsistencies, where code with similar logic implementing the same functionality across the arch files of two ISAs is not properly and concurrently updated.

Meanwhile, the cross-ISA participation rate offers further insight into the multifaceted expertise of participating developers. For example, LuaJIT has a cross-ISA participation rate of 100.0%,

indicating that all developers have made changes across multiple ISAs. A closer examination of this project revealed that all the arch files are maintained by a single developer. In contrast, libffi exhibits a cross-ISA participation rate of only 11.6%, suggesting that its ISA modules are relatively independent, with few developers working across different ISAs. In such cases, there is an increased risk of delayed or even omitted updates to functionally equivalent code across different ISAs.

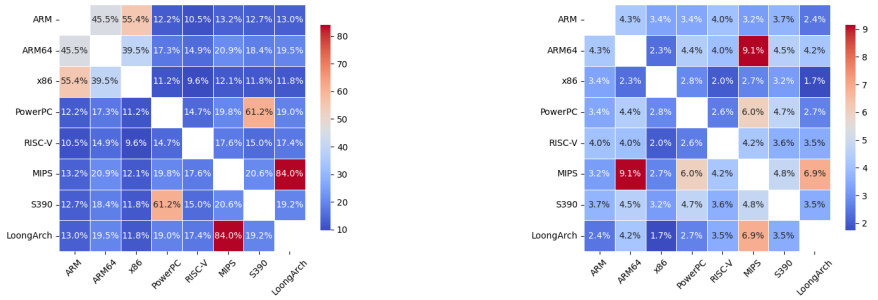


Fig. 9. The co-change rate between ISAs in V8 (left) and Linux (right).

Additionally, we computed the pairwise co-change rate between mirrored arch files. We also focus on the results of V8 and Linux projects shown in Figure 9. They exhibit relatively high (23.9%) and low (3.2%) project-level co-change rates, respectively. In the V8 project, the MIPS–LoongArch pair has an exceptionally high co-change rate of 84.0%, nearly every modification to the mirrored arch files occurs as a co-change. LoongArch was designed with historical compatibility with, and design influences from, MIPS, which can lead to shared or similarly structured implementation logic. We also find that the initial LoongArch implementation was adapted from MIPS. As a result, cross-ISA edits may be more likely when developers apply analogous fixes or synchronize behavior between MIPS and LoongArch. Likewise, the S390–PowerPC pair shows a relatively high co-change rate (61.2%). We observe that, in V8, the ISA pairs with high code similarity also tend to have high co-change rates.

By contrast, in the Linux project, the differences in co-change rates across ISAs are not pronounced, nor do they correlate strongly with inter-ISA similarity. In V8, the maintainers for MIPS and LoongArch appear to substantially overlap, which may increase the likelihood that a single change is applied to both targets within the same change set, leading to higher co-change. In contrast, in Linux these ISAs appear to be handled by different teams with separate review pipelines and patch flows. Consequently, changes may be submitted, reviewed, and merged more independently, which could contribute to the lower co-change we observe.

Answer to RQ3: Most projects clearly demonstrate co-change and cross-ISA participation across ISAs, which are especially prominent between certain ISA pairs, such as MIPS and LoongArch. The overall weighted average co-change rate is 7.2%, and the cross-ISA participation rate is 26.5%. Some of the projects have relatively close inter-ISA collaboration (Unicorn, LuaJIT), while others demonstrate a more independent cross-ISA development strategy (GCC, QEMU).

6 Discussion

We discuss our findings from three perspectives: support for multiple ISAs, management of similar code across ISAs, and potential tools to facilitate multi-ISA software development.

6.1 Support for Multiple ISAs

Our analysis of arch files systematically explains why distinct ISA-specific implementations are unavoidable in foundational software systems, due to differences in instruction semantics, calling conventions/ABI, and performance-critical hardware features. The roles of arch files can help practitioners make more informed engineering decisions about when distinct ISA-specific implementations are genuinely required, enabling them to better prioritize engineering and review effort and complement anecdotal experience with evidence.

The presence of eight frequently active ISAs in the studied 20 projects suggests that supporting a wide range of ISAs entails significant additional development effort. In total, we extracted 18,900 arch files, corresponding to 945 files per project. On average, each project contained 134 arch files per ISA. Development for each ISA requires specialized expertise and skills. The amount of development effort can increase substantially as the breadth of ISA support grows. Therefore, a software project should carefully consider which ISAs to support in order to manage costs effectively. This also highlights the need for a unified and efficient abstraction of ISAs to minimize the requirement for dedicated implementations.

6.2 Management of Similar Code across ISAs

Although separate implementations across ISAs are necessary, we find that 34.2% of arch files have mirrored counterparts, and that the overall weighted-average similarity across these mirrored pairs is 21.7%. The existence of similar code across ISAs within a project often results in redundant maintenance overhead and contributes to a higher probability of software defects. Although generally discouraged, developers often write similar code, whether intentionally, unintentionally, or due to unavoidable circumstances. We find that code across different ISAs displays a considerable degree of similarity. The average similarity of mirrored files in different projects ranges from 9.5% to 43.7%. This suggests potential duplicated effort and highlights the complexity of coordinated development and maintenance for mirrored arch files. Developers may need to pay closer attention to ISA-specific code changes outside their immediate scope. Collaborative design discussions could help refactor the arch files and eliminate unnecessary code clones.

Our investigation into the maintenance patterns of the mirrored arch files has revealed that, there is a certain degree of co-change and cross-ISA participation rate among mirrored arch files across different ISAs. We find 2,778 commits that simultaneously modified arch files of multiple ISAs. This suggests that cross-ISA updates can be coordinated and performed by individual developers within a single commit. When the same group of developers is responsible for multiple ISAs, cross-ISA updates may become more consistent. Cross-ISA expertise plays an important role in projects that support multiple ISAs. Cultivating more of these developers to build and sustain such expertise, through documentation, mentorship, and opportunities to review and modify ISA-specific code, may help reduce knowledge silos and improve the timeliness and consistency of cross-ISA maintenance.

Meanwhile, mirrored arch files also evolve in a loosely coupled manner, with counterpart updates occurring in different commits and at different times. Figure 10 presents an example of cross-ISA changes to mirrored arch files in V8 spanning three commits (*064b9a7903b793734b6c03a86ee53a2dc85f0f80*, *13192d6e10fa726858056e49fc9bca6201401171*, *cda104f32c0389c3535f218228c10f6fdbb0a59f*) by three different developers, where both the code and the corresponding modifications remain consistent across ARM64, ARM, x86, RISC-V, MIPS, and LoongArch. In such cases, functionally equivalent implementations may be more prone to untimely or inconsistent updates.

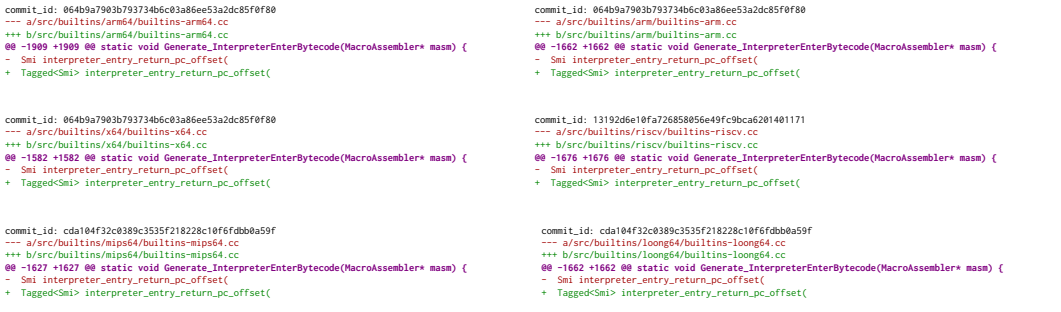


Fig. 10. An illustrative example of cross-ISA changes to mirrored arch files in V8 spanning multiple commits.

6.3 Automated Tools for Multi-ISA Development

Our empirical findings suggest a promising direction for improving the efficiency of multi-ISA development. Specifically, researchers and practitioners can develop automated tools to detect similar code fragments and recommend corresponding changes across ISAs. Our methodology can directly motivate and inform the development of practical tool support for maintaining mirrored arch files. In particular, our similarity computation can be used to identify candidate mirrored file pairs across ISAs, and derive a fine-grained correspondence between similar code regions. For example, the mirrored arch files for different ISAs shown in Figure 10 can be identified as a candidate set. Building on this correspondence, a tool can parse each arch file into an abstract syntax tree (AST) and represent a change to one file as an AST edit script, e.g., insertions, deletions, and updates. The edit script can then be mapped and propagated to the AST of the counterpart file, producing a candidate synchronized patch that developers can review and apply. For example, when the first commit in Figure 10 was made, the tool could have proactively recommended corresponding updates for the remaining mirrored files, potentially reducing the need for the follow-up modifications that were later applied in two separate commits.

7 Threats to Validity

7.1 Internal Validity

Threats to internal validity arise from three aspects: arch file role identification, mirrored arch file identification, and code similarity calculation.

This study examines the rationale and necessity for separate ISA implementations within projects by identifying the roles of arch files, rather than attempting to provide an exhaustive taxonomy of all possible reasons for requiring arch files. While not aiming for a fine-grained classification, interpreting the semantics of large-scale arch files remains challenging. To address this, three authors manually identified the roles in a sample set, drawing on their extensive software development experience and over three years of ISA expertise. We further validated the completeness of these roles by applying three large language models (LLMs) to all 18,900 arch files. The reliability of the LLMs was further assessed by manually reviewing a sample of their outputs. In addition, cloud-hosted LLM services may evolve over time, which can lead to variations in the results.

File structures in software systems typically reflect the functionalities of the files. The *arch/*-style directory layout, together with ISA-keyword patterns in paths and filenames, is a widely adopted organizational convention in the studied projects. Following this common development practice, we identify mirrored arch files based on their file paths and file names. Our scope excludes ISA-specific code fragments that are either outside mirrored ISA-specific directories or interleaved in generic files, e.g., `#ifdef`-guarded blocks. Notably, this study focuses on the mirrored-arch-file pattern.

We focus on the code similarity between mirrored arch files to gain insights into managing their commonalities. Conducting similarity calculations across all arch files would incur exponentially higher costs, while yielding only marginal additional benefits. The accuracy of similarity calculation depends on the chosen tool and can be sensitive to the underlying AST representation and matching algorithm. Accordingly, we carefully select GumTree based on the rationale provided in Section 4.4.2.

7.2 External Validity

Threats to external validity involve three aspects: the selection of studied projects, the choice of ISAs and the studied time window.

We collected a diverse set of projects known to contain ISA-specific code. In our dataset, which consists primarily of low-level foundational software projects, ISA-specific functionality is mainly implemented in C/C++. Therefore, our findings may not generalize to other project types or to projects where ISA-specific code is written in other languages. Although this set does not cover every possible project, it consists of widely adopted foundational systems that vary in domain, popularity, and scale. Because modern software stacks provide abstraction and shielding from hardware details, projects that directly involve ISA-specific code account for a relatively small proportion of the overall software landscape. This characteristic makes our dataset particularly valuable for examining ISA-related development practices, thereby enhancing the relevance and applicability of our findings.

Our study covers eight ISAs, including both widely used and actively maintained ISAs in the Linux kernel, as well as emerging ISAs such as RISC-V. This selection captures a broad spectrum of ISA development, thereby ensuring strong representativeness and reinforcing the validity of our analysis.

The temporal scope of our co-change and participation analysis is limited to 2022–2024. Observations may differ across development phases. Nevertheless, this window covers the bring-up of RISC-V support in projects such as V8 and FFmpeg, as well as the bring-up of LoongArch support in all projects that include it. The 2022–2024 window helps maintain consistent cross-project and cross-ISA alignment, as several ISAs were introduced relatively recently during this period. Although we did not conduct a longitudinal investigation, we performed a preliminary analysis of multiple V8 releases and found that the similarity remains relatively stable over time. A more extensive longitudinal study would be a valuable direction for future work.

8 Conclusion

In this study, we extracted the arch files associated with eight ISAs from 20 software projects. We identified mirrored arch files from the arch files and conducted a similarity analysis. Our findings provide empirical evidence and practical guidance for practices, methodologies, and tools for managing similar code in ISA-specific implementations. In future work, we plan to develop automated tools to suggest corresponding modifications across ISAs.

9 Data Availability

The data and scripts used in this study are available in the following GitHub repository: https://github.com/lxtqa/ISAsimilarity.

Acknowledgments

This work is supported by the Major Project of ISCAS (ISCAS-ZD-202302), and Youth Innovation Promotion Association of the Chinese Academy of Sciences (2023121).

References

[1] Akshat Agrawal and Sumit Kumar Yadav. 2013. A hybrid-token and textual based approach to find similar code segments. In *2013 fourth international conference on computing, communications and networking technologies (ICCCNT)*. IEEE, 1–4. doi:10.1109/icccnt.2013.6726700

[2] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A systematic review on code clone detection. *IEEE access* 7 (2019), 86121–86144. doi:10.1109/access.2019.2918202

[3] Ajmain I. Alam, Palash R. Roy, Farouq Al-Omari, Chanchal K. Roy, Banani Roy, and Kevin A. Schneider. 2023. GPTCloneBench: A comprehensive benchmark of semantic clones and cross-language clones using GPT-3 model and SemanticCloneBench. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 1–13. doi:10.1109/ICSME58846.2023.00013

[4] Pouria Alikhanifard and Nikolaos Tsantalis. 2025. A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–63. doi:10.1145/3696002

[5] Hela Belhadj Amor, Carolynn Bernier, and Zdeněk Přikryl. 2021. A RISC-V ISA extension for ultra-low power IoT wireless signal processing. *IEEE Trans. Comput.* 71, 4 (2021), 766–778. doi:10.1109/tc.2021.3063027

[6] Hugo Andrade and Ivica Crnkovic. 2018. A review on software architectures for heterogeneous platforms. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 209–218. doi:10.1109/apsec.2018.00035

[7] Hamid Abdul Basit and Stan Jarzabek. 2009. A data mining approach for detecting higher-level clones in software. *IEEE Transactions on Software engineering* 35, 4 (2009), 497–514. doi:10.1109/tse.2009.16

[8] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 368–377. doi:10.1109/ICSM.1998.738528

[9] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering* 33, 9 (2007), 577–591. doi:10.1109/tse.2007.70725

[10] Ben Boyter. 2025. *scc: v3.5.0*. https://github.com/boyter/scc

[11] Lutz Büch and Artur Andrzejak. 2019. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 95–104. doi:10.1109/saner.2019.8668039

[12] S Carter, RJ Frank, and DSW Tansley. 2013. Clone detection in telecommunications software systems: A neural net approach. In *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications*. Psychology Press, 273–280.

[13] Chang-Feng Chen, Azlan Mohd Zain, and Kai-Qing Zhou. 2022. Definition, approaches, and analysis of code duplication detection (2006–2020): a critical review. *Neural Computing and Applications* 34, 23 (2022), 20507–20537. doi:10.1007/s00521-022-07707-2

[14] Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, and Sergio Saponara. 2021. A lightweight posit processing unit for RISC-V processors in deep neural network applications. *IEEE transactions on emerging topics in computing* 10, 4 (2021), 1898–1908. doi:10.1109/tetc.2021.3120538

[15] Georgina Cosma and Mike Joy. 2011. An approach to source-code plagiarism detection and investigation using latent semantic analysis. *IEEE transactions on computers* 61, 3 (2011), 379–394. doi:10.1109/tc.2011.223

[16] Emilio G Cota and Luca P Carloni. 2019. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 74–87. doi:10.1145/3313808.3313811

[17] Neil Davey, Paul Barson, Simon Field, Ray Frank, and D Tansley. 1995. The development of a software clone detector. *International Journal of Applied Software Technology* 1, 3/4 (1995), 219–236.

[18] Jean-Rémy Falleri and Matias Martinez. 2024. Fine-grained, accurate and scalable source differencing. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 231:1–231:12. doi:10.1145/3597503.3639148

[19] Chen Gao, Xiangwei Meng, Wei Li, Jinhui Lai, Yiran Zhang, and Fengyuan Ren. 2024. {CrossMapping}: Harmonizing Memory Consistency in {Cross-ISA} Binary Translation. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 1013–1028.

[20] Muhammad Hammad, Önder Babur, Hamid Abdul Basit, and Mark van den Brand. 2020. Deepclone: modeling clones to generate code predictions. In *Reuse in Emerging Software Engineering Practices: 19th International Conference on Software and Systems Reuse, ICSR 2020, Hammamet, Tunisia, December 2–4, 2020, Proceedings 19*. Springer, 135–151. doi:10.1007/978-3-030-64694-3_9

[21] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier. doi:10.1016/0026-2692(93)90111-q

[22] Jinhu Jiang, Chaoyi Liang, Rongchao Dong, Zhaohui Yang, Zhongjun Zhou, Wenwen Wang, Pen-Chung Yew, and Weihua Zhang. 2024. A System-Level Dynamic Binary Translator Using Automatically-Learned Translation Rules. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 423–434. doi:10.1109/cgo57630.2024.10444850

[23] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105. doi:10.1109/icse.2007.30

[24] Haochen Jin, Zhanqi Cui, Shifan Liu, and Liwei Zheng. 2022. Improving code clone detection accuracy and efficiency based on code complexity analysis. In *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 64–72. doi:10.1109/dsa56465.2022.00017

[25] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering* 28, 7 (2002), 654–670. doi:10.1109/tse.2002.1019480

[26] S Karthik and B Rajdeepa. 2022. A collaborative method for code clone detection using a deep learning model. *Advances in Engineering Software* 174 (2022), 103327. doi:10.1016/j.advengsoft.2022.103327

[27] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. 187–196. doi:10.1145/1095430.1081737

[28] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone detection using abstract syntax suffix trees. In *2006 13th Working conference on reverse engineering*. IEEE, 253–262. doi:10.1109/wcre.2006.18

[29] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings eighth working conference on reverse engineering*. IEEE, 301–309. doi:10.1109/wcre.2001.957835

[30] Jens Krinke. 2007. A study of consistent and inconsistent changes to code clones. In *14th working conference on reverse engineering (WCRE 2007)*. IEEE, 170–178. doi:10.1109/wcre.2007.7

[31] Arun Lakhotia, Junwei Li, Andrew Walenstein, and Yun Yang. 2003. Towards a clone detection benchmark suite and results archive. In *11th IEEE International Workshop on Program Comprehension, 2003.* IEEE, 285–286.

[32] Maggie Lei, Hao Li, Ji Li, Namrata Aundhkar, and Dae-Kyoo Kim. 2022. Deep learning application on code clone detection: A review of current knowledge. *Journal of Systems and Software* 184 (2022), 111141. doi:10.1016/j.jss.2021.111141

[33] Francesco Leone and Shingo Takada. 2022. Towards overcoming type limitations in semantic clone detection. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*. IEEE, 25–31. doi:10.1109/iwsc55060.2022.00013

[34] Guangjie Li, Hui Liu, Yanjie Jiang, and Jiahao Jin. 2018. Test-based clone detection: an initial try on semantically equivalent methods. *IEEE access* 6 (2018), 77643–77655. doi:10.1109/access.2018.2883699

[35] Jia Li, Chongyang Tao, Zhi Jin, Fang Liu, Jia Li, and Ge Li. 2024. ZC3: Zero-Shot Cross-Language Code Clone Detection. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, 875–887. doi:10.1109/ASE56229.2023.00210

[36] Yuancheng Li, Chaohang Yu, and Yaqi Cui. 2023. TPCaps: A framework for code clone detection and localization based on improved CapsNet. *Applied Intelligence* 53, 13 (2023), 16594–16605. doi:10.1007/s10489-022-03158-3

[37] Alexandre Lopoukhine, Federico Ficarelli, Christos Vasiladiotis, Anton Lydike, Josse Van Delm, Alban Dutilleul, Luca Benini, Marian Verhelst, and Tobias Grosser. 2025. A multi-level compiler backend for accelerated micro-kernels targeting risc-v isa extensions. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 163–178. doi:10.1145/3696443.3708952

[38] Mayrand, Leblanc, and Merlo. 1996. Experiment on the automatic detection of function clones in a software system using metrics. In *1996 Proceedings of International Conference on Software Maintenance*. IEEE, 244–253. doi:10.1109/icsm.1996.565012

[39] Hou Min and Zhang Li Ping. 2019. Survey on software clone detection research. In *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences*. 9–16. doi:10.1145/3312662.3312707

[40] Gilad Mishne, Maarten De Rijke, et al. 2004. Source Code Retrieval using Conceptual Similarity.. In *RIAO*, Vol. 4. 539–554.

[41] Matija Novak, Mike Joy, and Dragutin Kermek. 2019. Source-code similarity detection and detection tools used in academia: a systematic review. *ACM Transactions on Computing Education (TOCE)* 19, 3 (2019), 1–37. doi:10.1145/3313290

[42] Joel Ossher, Hitesh Sajnani, and Cristina Lopes. 2011. File cloning in open source java projects: The good, the bad, and the ugly. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 283–292. doi:10.1109/icsm.2011.6080795

[43] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information Needs in Contemporary Code Review. *Proc. ACM Hum. Comput. Interact.* 2, CSCW (2018), 135:1–135:27. doi:10.1145/3274404

[44] Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. 2002. Finding plagiarisms among a set of programs with JPlag. *J. Univers. Comput. Sci.* 8, 11 (2002), 1016.
[45] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199. doi:10.1016/j.infsof.2013.01.008
[46] Viktor Razilov, Emil Matúš, and Gerhard Fettweis. 2022. Communications signal processing using RISC-V vector extension. In *2022 International Wireless Communications and Mobile Computing (IWCMC)*. IEEE, 690–695. doi:10.1109/iwcmc55113.2022.9824961
[47] Iris Reinhartz-Berger and Anna Zamansky. 2020. Reuse of similarly behaving software through polymorphism-inspired variability mechanisms. *IEEE Transactions on Software Engineering* 48, 3 (2020), 773–785. doi:10.1109/tse.2020.3001512
[48] Matthew Renze. 2024. The effect of sampling temperature on problem solving in large language models. In *Findings of the association for computational linguistics: EMNLP 2024*. 7346–7356. doi:10.18653/v1/2024.findings-emnlp.432
[49] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of computing TR* 541, 115 (2007), 64–68.
[50] Chanchal K Roy and James R Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th iEEE international conference on program comprehension*. IEEE, 172–181. doi:10.1109/icpc.2008.41
[51] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering*. 1157–1168. doi:10.1007/978-981-16-1927-4_4
[52] Thomas Schmorleiz and Ralf Lämmel. 2016. Similarity management of'cloned and owned'variants. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. 1466–1471. doi:10.1145/2851613.2851785
[53] Gehan MK Selim, King Chun Foo, and Ying Zou. 2010. Enhancing source-based clone detection using intermediate representation. In *2010 17th working conference on reverse engineering*. IEEE, 227–236. doi:10.1109/wcre.2010.33
[54] Temple F Smith, Michael S Waterman, et al. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197. doi:10.1016/0022-2836(81)90087-5
[55] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. 2016. Identifying functionally similar code in complex codebases. In *2016 ieee 24th international conference on program comprehension (icpc)*. IEEE, 1–10. doi:10.1109/icpc.2016.7503720
[56] Jeffrey Svajlenko and Chanchal K Roy. 2015. Evaluating clone detection tools with bigclonebench. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 131–140. doi:10.1109/icsm.2015.7332459
[57] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. 2010. An empirical study on the maintenance of source code clones. *Empirical Software Engineering* 15, 1 (2010), 1–34. doi:10.1007/s10664-009-9108-x
[58] Fabien Patrick Viertel, Wasja Brunotte, Daniel Strüber, and Kurt Schneider. 2019. Detecting Security Vulnerabilities using Clone Detection and Community Knowledge.. In *SEKE*. 245–324. doi:10.18293/seke2019-183
[59] Min Wang, Pengcheng Wang, and Yun Xu. 2017. CCSharp: An efficient three-phase code clone detector using modified PDGs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 100–109. doi:10.1109/apsec.2017.16
[60] Shihang Wang, Jianghan Zhu, Qi Wang, Can He, and Terry Tao Ye. 2021. Customized instruction on RISC-V for Winograd-based convolution acceleration. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 65–68. doi:10.1109/asap52443.2021.00018
[61] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. 2018. Enhancing cross-isa dbt through automatically learned translation rules. *ACM SIGPLAN Notices* 53, 2 (2018), 84–97. doi:10.1145/3296957.3177160
[62] Xi Wang, John D Leidel, Brody Williams, Alan Ehret, Miguel Mark, Michel A Kinsy, and Yong Chen. 2021. xbgas: A global address space extension on risc-v for high performance computing. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 454–463. doi:10.1109/ipdps49936.2021.00054
[63] Yafang Wang and Dongsheng Liu. 2019. Image-based clone code detection and visualization. In *2019 International Conference on Artificial Intelligence and Advanced Manufacturing (AIAM)*. IEEE, 168–175. doi:10.1109/aiam48774.2019.00041
[64] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 87–98. doi:10.1145/2970276.2970326
[65] Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. 2023. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software* 204 (2023), 111796. doi:10.1016/j.jss.2023.111796
[66] Aiping Zhang, Liming Fang, Chunpeng Ge, Piji Li, and Zhe Liu. 2023. Efficient transformer with code token learner for code clone detection. *Journal of Systems and Software* 197 (2023), 111557. doi:10.1016/j.jss.2022.111557
[67] Ziyi Zhao, Zhang Jiang, Ying Chen, Xiaoli Gong, Wenwen Wang, and Pen-Chung Yew. 2021. Enhancing atomic instruction emulation for cross-isa dynamic binary translation. In *2021 IEEE/ACM International Symposium on Code*

Generation and Optimization (CGO). IEEE, 351–362. doi:10.1109/cgo51591.2021.9370312

Received 2026-02-25; accepted 2026-03-24