

Article

AIPR: An Automated Instruction-Level Patching and Rewriting Framework for Sustainable RISC-V Research

Juhee Choi

Department of Smart Information and Telecommunications Engineering, Sangmyung University, Cheon-An, Cheonan-si 31066, Republic of Korea; jhplus@smu.ac.kr

Abstract

Computer systems research faces significant challenges in reproducibility because of toolchain fragmentation and the rapid evolution of the RISC-V ecosystem. Many research artifacts stay as ‘digital tombstones’ because they lack stable build environments and suffer from undocumented dependencies. This work presents the AIPR (Automated Instruction-level Patching and Rewriting) framework to address the gap between unstable hardware specifications and reproducible research. The methodology shifts the focus from complex source-level recompilation to direct executable-level modification. A three-stage pipeline automates instruction-level analysis, immediate reconstruction, and binary patching in ELF binaries. Experimental evaluations with the V-FRONT RISC-V processor include 2000 independent trials. These trials verify the functional robustness of the framework under complex architectural constraints. Furthermore, the AIPR framework achieves a $29.57\times$ speedup in artifact generation compared to traditional GCC-based flows.

Keywords: binary rewriting; reproducibility; toolchain fragmentation; RISC-V

1. Introduction

In modern computer architecture and systems research, experimental verification is a fundamental methodology [1,2]. Most researchers validate their new architectural ideas using software-based simulators or FPGA-based prototypes [3–6]. This heavy reliance on digital artifacts leads to a greater emphasis on tool accessibility. Consequently, open-source platforms such as GitHub and GitLab facilitate the growth of ‘open science’ [7,8]. These environments provide the infrastructure for code sharing and collaborative research.

However, reproducing these results is often difficult in practice [9–11]. Code repositories in published papers often disappear over time [12]. Some repositories stay only as ‘digital tombstones,’ and these are artifacts that exist but researchers cannot build or execute them anymore. Even with available source code, researchers face obstacles such as undocumented dependencies, old toolchains, and hidden environmental assumptions [13].

This reproducibility gap was systematically quantified by Collberg et al., who examined the repeatability of computer systems research [9]. Their results revealed that fewer than one-third of surveyed papers could be successfully rebuilt within a realistic reuse window. These findings show a major problem in systems research; software artifacts are often treated as temporary tools instead of long-term scientific results [14].

The issue is especially serious in computer architecture research because experimental pipelines depend on complex toolchains [3,15]. Small changes in instruction set specifications, compilers, or simulators can render years of prior work unusable. Accordingly, researchers spend significant time just managing these environments instead of focusing



Academic Editor: Andrea Prati

Received: 12 January 2026

Revised: 28 January 2026

Accepted: 29 January 2026

Published: 31 January 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

on new architectural ideas. Modifying executable files is more practical than recompiling source code, which often fails due to toolchain issues, because it avoids a full system rebuild [16,17].

Based on these observations, this work is guided by the following research question:

- **RQ:** Can executable-level binary rewriting effectively mitigate reproducibility and sustainability challenges caused by toolchain fragmentation in RISC-V systems research, while maintaining functional robustness and performance efficiency?

Proposed strategies involve a shift in the intervention point from the source level to the executable level. AIPR (Automated Instruction-level Patching and Rewriting) serves as a binary-level transformation framework. The key advantage of AIPR revives the reusability of research artifacts even when reproduction remains impossible due to broken environments. By analyzing and rewriting instructions within compiled binaries, the framework allows for both rapid iteration and the continued use of legacy machine code streams. This process keeps architectural correctness and execution results without needing the original source code [18,19].

The main contributions of this work are summarized as follows:

- **Executable-Level Methodology:** This work shifts the experimental intervention point from fragile source-level recompilation to stable executable-level modification to improve research reproducibility under toolchain fragmentation.
- **AIPR Framework:** The AIPR framework automates instruction-level analysis, immediate reconstruction, and direct binary patching within ELF binaries.
- **Instruction Encoding Automation:** A specialized encoding engine ensures correct split-immediate recalculation and sign-extension handling during binary rewriting.
- **Accelerated Verification:** The proposed approach achieves a $29.57\times$ speedup in artifact generation compared to GCC-based recompilation, significantly reducing verification turnaround time.

The rest of this paper is organized as follows. Section 2 reviews the background and related challenges underlying the proposed framework. Section 3 details the overall architecture and transactional workflow of the AIPR framework. The reliability and performance of the proposed framework are evaluated in Section 4. Finally, Section 5 concludes the work and suggests future research.

2. Background

2.1. Structural Barriers to Reproducibility in Systems Research

Reproducibility has become a persistent barrier in modern computer systems research as experimental pipelines increase in scale and complexity. Many studies depend on large and fragile software toolchains, and long-term reuse of research artifacts remains difficult even when the original methodology is sound. This challenge appears clearly in widely used infrastructures such as the gem5 simulator, which models detailed hardware behavior and relies on both C++ implementations and Python3.11-based configuration layers [3]. The gem5 simulator is a common platform for architectural studies. However, frequent updates often prevent older research from running in newer environments.

Comparable reproducibility obstacles also arise in the RISC-V ecosystem, where rapid specification updates and extension revisions are common [20–22]. Extensibility represents a core strength of RISC-V, but incompatible toolchain versions and changes in compiler or kernel support regularly invalidate existing experimental setups. Researchers often face a choice between maintaining outdated environments or discontinuing follow-on studies. These structural difficulties reveal a structural gap between short-term experimental success and long-term reproducibility. This environment fosters the emergence of the Research

Software Engineer (RSE) role, which applies software engineering practices to sustain executable research artifacts over time [14,23].

2.2. Binary Rewriting as an Alternative Intervention Point

Binary rewriting provides a practical way to modify executable files when source code is unavailable [16,17]. Researchers previously focused on security hardening and performance profiling with such techniques, but the scope now expands to general systems research. The process requires an analysis of control-flow structures and instruction encodings to ensure safe code insertion or replacement.

Recent studies improve binary-level transformation reliability and allow stable modifications with low overhead, even in complex execution environments [16,18]. Direct intervention at the executable level avoids the fragilities of source-level toolchains, so such an approach preserves program behavior and prevents build-time errors.

Such features establish binary rewriting as a strong base for architectural experiments because executable-level manipulation separates logic from unstable build environments [12]. Systematic isolation of the experimental layer ensures a reliable foundation for long-term research artifacts, and such benefits led to the AIPR framework design in this study.

3. AIPR Framework

3.1. AIPR Framework Implementation

The AIPR framework, implemented in Python, serves as a specialized engine for instruction-level analysis, immediate reconstruction, and direct binary patching. The system orchestrates a three-stage pipeline to transform high-level architectural parameters into physical binary modifications:

- **ELF Parsing and Section Discovery:** The engine parses the section header table to identify the offset and size of the .text section. This stage remains a prerequisite to ensure that all modifications occur strictly within executable boundaries.
- **Instruction Injection and Encoding:** Assembly patterns undergo conversion into 32-bit or 16-bit hexadecimal streams. This process requires a precise implementation of RISC-V immediate encoding logic [24,25].
- **Direct Binary Patching:** The framework overwrites the target byte stream at specific offsets. The current version excludes any modifications to the ELF header. This approach ensures file integrity and simulator compatibility. It also avoids complex structural recalculations.

The technical analysis focuses primarily on the Instruction Encoding Engine and the Binary Patching Engine. These modules manage the core algorithmic complexity, including immediate reconstruction and atomic hexadecimal stream manipulation. In contrast, the initial parsing stage utilizes standard structural analysis, so it performs a secondary role. Tables 1 and 2 present simplified Python-like pseudocode for clarity. The production implementation includes additional logic for relocation-aware offset resolution and transactional verification.

The implementation of the Instruction Encoding Engine, as detailed in Table 1, provides a high-level abstraction layer that isolates the developer from bit-level architectural specifics. The engine encapsulates complex bit-masking and shifting operations within specialized functions such as `encode_u_type` and `encode_i_type`. Such encapsulation ensures that architectural parameters map accurately to specific field positions—including `rd`, `rs1`, and `funct3`—defined by the RISC-V ISA. Such modularity enables the framework to process 32-bit constants as abstract input variables. The engine then decomposes these values into valid instruction formats and eliminates the need for manual bit-field calcula-

tions. Furthermore, the `generate_binary_pair` API serves as a critical orchestration point that maintains structural consistency throughout the encoding workflow. The method manages the exact sequence of instruction generation and guarantees that the LUI and ADDI pairs couple correctly.

Table 1. Pseudocode of Instruction Encoding Engine for Assembly-to-Binary Transformation.

```

def calculate_riscv_immediates(self, target_value):
    low_12 = target_value & 0xFFF
    up_20 = (target_value >> 12) & 0xFFFFF
    # Compensation for RISC-V ADDI sign-extension
    if low_12 >= 0x800:
        up_20 = (up_20 + 1) & 0xFFFFF
    return up_20, low_12

def encode_u_type(self, imm_20, opcode=0x37):
    # Format: imm[31:12] | rd[11:7] | opcode[6:0]
    instr = (imm_20 << 12) | (self.rd << 7) | opcode
    return instr

def encode_i_type(self, imm_12, rs1=None, funct3=0x0, opcode=0x13):
    # Format: imm[11:0] | rs1[19:15] | funct3[14:12] | rd[11:7] | opcode[6:0]
    if rs1 is None: rs1 = self.rd # Default to self-increment
    instr = (imm_12 << 20) | (rs1 << 15) | (funct3 << 12) | (self.rd << 7) |
        opcode
    return instr

def generate_binary_pair(self, new_param):
    up, low = self.calculate_riscv_immediates(new_param)
    lui_bin = self.encode_u_type(up)
    addi_bin = self.encode_i_type(low)
    return lui_bin, addi_bin

```

The Binary Rewriting Engine (Table 2) adopts a state-managed approach to file manipulation and ensures stability during the rewriting process. The engine maintains a local buffer of `binary_data` and utilizes the `r+b` (read-plus-binary) file mode to perform targeted seek operations. The chosen strategy significantly minimizes disk I/O overhead. Localized manipulation remains essential for the preservation of ELF structure integrity, as the engine modifies specific byte-ranges in-place. The technique removes the necessity for full binary re-serialization and prevents unintended alignment or offset shifts in non-target sections.

Operational reliability stems from the `execute_automated_flow` method, which functions as a centralized control loop with rigorous exception handling. The orchestration logic treats the entire patching sequence from pattern discovery via `find_instruction_offset` to the final write as a single transactional unit. If the framework fails to locate the target pattern or if the write operation encounters an error, the engine immediately interrupts the workflow. The mechanism protects mission-critical firmware or simulated hardware binaries from the execution of partially patched or corrupted machine code.

Table 2. Pseudocode of Binary Rewriting Engine for Direct Hexadecimal Stream Manipulation.

```

def find_instruction_offset(self, original_hex_sequence):
    target_bytes = bytes.fromhex(original_hex_sequence)
    offset = self.binary_data.find(target_bytes)
    if offset == -1:
        raise ValueError("[AIPR_Error]Target_instruction_pattern_not_found.")
    return offset

def apply_hex_patch(self, offset, patched_hex_sequence):
    patch_bytes = bytes.fromhex(patched_hex_sequence)
    with open(self.file_path, 'r+b') as f:
        f.seek(offset)
        f.write(patch_bytes)
        f.flush()
        # Verification Logic
        f.seek(offset)
        if f.read(len(patch_bytes)) != patch_bytes:
            raise IOError("[AIPR_Error]Patch_verification_failed.")

def execute_automated_flow(self, search_pattern, replace_pattern):
    self.load_binary()
    try:
        target_offset = self.find_instruction_offset(search_pattern)
        self.apply_hex_patch(target_offset, replace_pattern)
    except Exception as e:
        print(f"Workflow_Interrupted:{e}")

```

3.2. Operational Methodology

The AIPR framework transforms architectural parameters into binary modifications. The system analyzes how hardware configurations are converted into low-level instructions. This methodology identifies how the source code defines registers and hardware units. Such analysis allows the framework to target specific parameters for modification at the executable level.

The RISC-V architecture uses instructions with a fixed length of 32 bits. Consequently, the system builds large addresses and constants through multiple steps. The framework tracks the synchronization of instruction pairs to maintain structural consistency. This tracking ensures that register base pointers stay correct when the system modifies hardware units across different memory regions.

Managing sign-extension is a major part of this process. The lower part of an instruction is treated as a signed integer, which can change the final value. The Instruction Encoding Engine calculates these limits in advance. The engine adjusts the upper instruction to keep the correct value even when the numbers are large.

The framework changes specific bit-fields in the machine code. The logic modifies only the immediate fields and leaves opcodes or register identifiers unchanged. This method ensures that the internal data flow stays the same. The approach avoids side effects for other hardware logic and keeps the program structure stable.

The final step modifies the hexadecimal data in the binary file. This process follows the little-endian memory layout of the architecture. The methodology also maintains alignment for different instruction lengths. Readers can find detailed tables and assembly sequences in the Appendix A.

4. Experimental Results and Analysis

This section presents an empirical evaluation designed to directly address the research question posed in Introduction. Specifically, the experiments assess whether executable-level binary rewriting can (i) maintain functional robustness across diverse instruction-level transformation challenges and (ii) significantly improve the efficiency of research artifact generation under fragmented toolchain conditions.

4.1. Experimental Environments

The primary objective of the proposed framework is to facilitate operational functionality in broken environments where source-level recompilation is practically unfeasible. Consequently, the fundamental focus of the experimental evaluation is to demonstrate the transition from technical infeasibility to feasibility. Detailed execution logs provide the evidence to determine whether the transformation is possible or impossible.

To supplement this functional proof-of-concept with quantitative data, additional empirical evaluations were conducted to measure performance efficiency. The time required to generate 20 unique test binaries was measured. The experimental environment consisted of a workstation running Ubuntu 22.04 LTS, equipped with an Intel Core i7-12700K processor (3.60 GHz) and 32 GB of DDR4 RAM. The evaluation compared the proposed framework against a traditional GCC-based recompilation flow.

To evaluate the AIPR framework in a realistic yet controlled architectural setting, all experiments utilize V-FRONT [26] as a representative baseline for verifying instruction-level transformations, as it strictly adheres to the RV32I v2.1 base integer instruction set. The most significant feature and constraint of this core is the combination of a Von Neumann structure and a 5-stage pipeline. The implementation incorporates MTE functionality through RTL modifications. However, the details of the hardware implementation remain beyond the scope of this paper, and the study excludes further technical discussion regarding the RTL changes. The detailed information is provided in Appendix B.

4.2. Functional Robustness and Reliability Validation

The initial phase of the evaluation focuses on the fundamental reliability across diverse architectural artifacts. This experiment serves as a formal verification of the technical feasibility of binary-level intervention. Unlike general architectural benchmarks that measure workload performance, the target binaries in this study are categorized by the structural obstacles they present to the rewriting process. Such structural challenges include standard immediate replacement, arithmetic correction for sign-extension boundaries, base address reconstruction, and global sequence propagation across dependent instructions.

To ensure the comprehensive robustness of the framework, the evaluation involved 2000 independent transformation trials, which were systematically categorized into four specialized challenge groups to ensure diversity: (1) isolated immediate replacement, (2) arithmetic sign-extension correction, (3) base address reconstruction, and (4) global sequence propagation. This confirms the framework's ability to handle various architectural patterns beyond simple variations. As shown in Table 3, for each of the 20 target binaries, 100 heterogeneous patching scenarios were executed.

As illustrated in Figure 1, the AIPR framework achieved a 100% success rate across all 2000 trials. Even in complex scenarios such as Type 2 arithmetic correction and Type 4 sequence propagation, no instances of binary corruption occurred. The evaluation follows a strict validation process to ensure that the patch logic preserves the original program semantics while the ELF header remains completely untouched. The procedure involves a three-tier check: structural consistency via disassembly, execution correctness through cycle-accurate RTL simulation, and control-flow integrity via program counter trace comparison.

Such a rigorous assessment confirms that the framework successfully updates security policies at the executable level and avoids any unintended functional divergence.

Table 3. Classification of Target Binaries based on Binary Rewriting Complexity and Architectural Dependencies.

Category	ID Range	Overwriting-Specific Challenges
Type 1	B1–B5	Standard Immediate Replacement: The engine updates isolated 12-bit or 20-bit immediate fields within a single instruction where no arithmetic dependencies exist.
Type 2	B6–B10	Arithmetic Correction: The process handles the 0x800 sign-extension boundary, which requires a simultaneous update of LUI and ADDI pairs to prevent value corruption.
Type 3	B11–B15	Base Address Modification: The framework targets the reconstruction of 32-bit memory-mapped register addresses through multi-instruction sequence analysis and offset alignment.
Type 4	B16–B20	Sequence Propagation: The system manages the consistent update of multiple dependent instructions across different hardware units to ensure global policy enforcement.

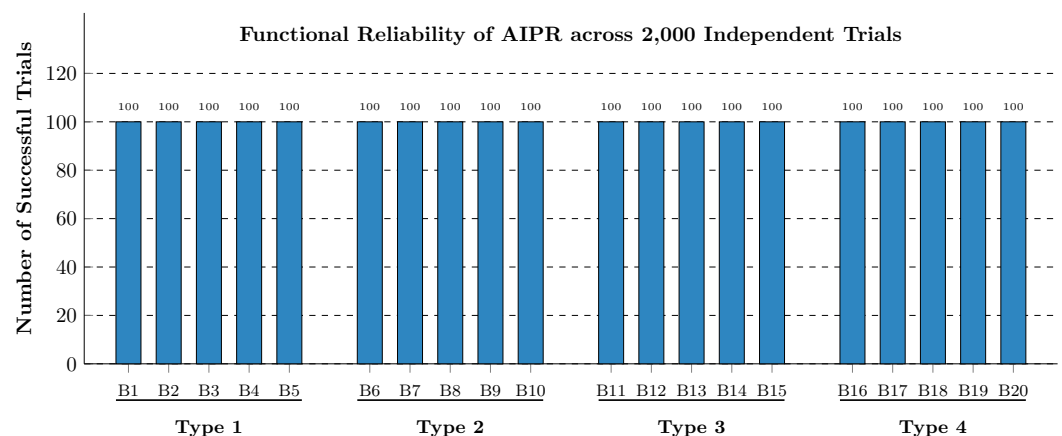


Figure 1. Functional Reliability of AIPR across 2000 Independent Trials. The framework achieved a 100% success rate across all four specialized groups (1–4).

4.3. Turnaround Time Comparison and Scalability Analysis

While the previous experiment validates the functional reliability, Table 4 evaluates its operational efficiency in accelerating the hardware verification loop. To provide a direct comparison with the functional validation, the turnaround time was measured across the same 2000 independent transformation trials (100 variants for each of the 20 target binaries across Types 1–4). The evaluation compared the cumulative time required to generate these artifacts using two distinct methodologies:

- **Baseline (Recompilation):** Modifying source-level parameters followed by a full toolchain execution (`riscv32-unknown-elf-gcc`) including parsing, optimization, and linking for each variant.
- **AIPR (Binary Patching):** Utilizing the proposed framework to directly manipulate immediates and opcodes within the pre-compiled artifacts.

The results indicate a significant $29.57\times$ speedup in artifact generation. In contrast, the traditional recompilation flow took 1.89 h for the 2000-trial campaign. This delay occurred because standard compilers use slow processes like multi-pass optimization and

symbol resolution. The AIPR framework now demonstrated that it restores research feasibility in only 230 s. The same task previously required nearly two hours of expert-level manual intervention or remained technically infeasible. This framework achieved a $29.57\times$ speedup, but this efficiency is a secondary benefit. The primary contribution is research sustainability in fragmented RISC-V ecosystems. The framework converts a multi-hour engineering struggle into a sub-four-minute automated task. This transition allows researchers and practitioners to reproduce artifacts efficiently. Consequently, the tool establishes a stable foundation for long-term research within the specialized architectural community.

Table 4. Quantitative Performance Comparison ($N = 2000$ Trials).

Metric	Recompilation Flow	AIPR Framework
Total Execution Time	~ 6800 s (~ 1.89 h)	230 s ($29.57\times$)
Average Time per Trial	3.4 s	0.115 s
Peak CPU Load	High (Multi-threaded)	Negligible
Disk I/O Intensity	High (Object Files)	Low (In-place)

4.4. Limitations and Comparative Analysis

The observed $29.57\times$ speedup demonstrated that instruction-level patching was highly efficient compared to the traditional GCC recompilation flow. I recognized that this gain was expected because the AIPR framework avoided the time-consuming overhead of multi-pass optimization and symbol resolution required by a compiler. However, the primary value of my proposal was the restoration of research feasibility in “broken” environments where source-level reconstruction was no longer possible. Unlike general binary rewriting frameworks that primarily targeted security hardening or performance profiling, the AIPR framework was a specialized methodology for architectural research sustainability. The framework separated experimental logic from unstable build environments to provide a stable foundation for long-term research artifacts.

Despite these advantages, the framework had specific failure cases. Because the engine excluded modifications to the ELF header to ensure simulator compatibility, the current system was unable to handle patches that required expanding binary sections beyond their original boundaries. Furthermore, complex structural changes to the binary file remained outside its capabilities. While this approach ensured high functional reliability for parameter modification, this design choice restricted the framework’s use for general-purpose code expansion or complex binary transformations.

The Instruction Encoding Engine is designed with a modular architecture that encapsulates bit-masking and shifting operations. I have explicitly noted that the framework can be extended to architectures like ARM or x86 by updating the encoding logic for their respective instruction formats. However, the AIPR framework utilizes a transactional, in-place patching method. The engine avoids modifications to the ELF header to maintain strict compatibility with architectural simulators. Because of this design, the framework cannot expand binary sections beyond their original boundaries. The current system is optimized for statically linked research artifacts. Additionally, it does not support complex formats such as position-independent executables (PIE) or dynamic linking. These formats are excluded because they require intensive structural recalculations. Moreover, aggressive compiler optimizations remain outside the current scope to prioritize semantic integrity. Such constraints limit practical applications in specific scenarios. For instance, researchers cannot currently insert large-scale security instrumentation or recalculate complex branch targets for long-distance jumps. Future research could address these issues with a full

ELF rebuilding engine. This proposed engine would enable section relocation and more extensive structural changes.

5. Conclusions and Future Work

This research addresses systemic inefficiencies in computer architecture research by proposing the AIPR framework. The system bridged the gap between volatile specifications and reproducible research. The framework shifted the intervention point from brittle source-level recompilation to the stable executable level. The evaluations prove that reliable binary-level modification is possible. This remains true despite constraints like the little-endian layout and sign-extension. The framework also handles instruction length variability in the RISC-V ISA. Through a transactional workflow, the engine ensured semantic integrity without the original source code. Experimental results validate the robustness of the approach across 2000 trials, where the AIPR framework delivered a $29.57\times$ speedup compared to traditional recompilation. The framework reduced the total execution time from 1.89 h to 230 s. Furthermore, the in-place manipulation maintained a negligible CPU load and low disk I/O, which proved the efficiency of the proposed binary rewriting methodology.

Future research for the AIPR framework will prioritize expanding verification coverage by generating non-standard binary layouts that standard compilers typically avoid. I aim to develop an architectural fuzzer and a Boundary-Seeker module to expose deep-seated microarchitectural errors by forcing illegal opcodes and branch instructions across critical hardware limits. Furthermore, I plan to explore the application of global-local feature fusion to accurately classify complex hardware-integrity anomalies within these generated traces, leveraging few-shot learning to identify rare bug signatures even with limited labeled data [27]. Finally, I will establish a roadmap for long-term sustainability by integrating AIPR into automated research workflows, focusing on concrete scenarios such as restoring legacy “digital tombstones” to demonstrate practical artifact reuse across evolving RISC-V specifications.

Funding: This research was funded by a 2023 research Grant from Sangmyung University(2023-A000-0100 and 2024-A000-0100).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data supporting the findings of this study are not publicly available due to restrictions on their disclosure; however, the experimental methodology and evaluation procedures are fully described in the manuscript.

Conflicts of Interest: The author declares no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

AIPR	Automated Instruction-level Patching and Rewriting
RSE	Research Software Engineer
ISA	Instruction Set Architecture
ELF	Executable and Linkable Format
RTL	Register Transfer Level
RVC	RISC-V Compressed Instruction Extension

LUI	Load Upper Immediate
ADDI	Add Immediate
SFR	Special Function Register
MMIO	Memory-Mapped I/O
CSR	Control and Status Register
LE	Little-Endian Representation

Appendix A. Case Study: RISC-V Instruction-Level Transformation

To demonstrate the practical application of the AIPR framework, this section explores how high-level C declarations for hardware control are translated into optimized RISC-V assembly and subsequently transformed at the binary level.

Appendix A.1. C-Based Hardware Register Initialization Example

The code presented in Table A1 illustrates a typical embedded system example of configuring the Memory Tag Extension (MTE)—a critical security feature—of each processor core in a multi-core architecture. In embedded system design, controlling hardware resources requires a precise definition of the physical address map where peripherals and core control registers reside. This example utilizes specific memory regions allocated during system bus design to independently control the security attributes of each processor core.

Table A1. Correspondence between C-Level Memory Mapped Configuration and RISC-V Assembly with Split-Immediate Reconstruction.

1. C Code for Testing
<pre> ... #define BASE_ADDR_SFR_MAP_CORE_0 0x2000f000 #define BASE_ADDR_SFR_MAP_CORE_1 0x20010000 #define BASE_ADDR_SFR_MAP_CORE_2 0x20011000 #define MEMORY_TAG_EXT_CFG_CORE_0 (BASE_ADDR_SFR_MAP_CORE_0 + 0x0c30) #define MEMORY_TAG_EXT_CFG_CORE_1 (BASE_ADDR_SFR_MAP_CORE_1 + 0x0c30) #define MEMORY_TAG_EXT_CFG_CORE_2 (BASE_ADDR_SFR_MAP_CORE_2 + 0x0c30) REG32(MEMORY_TAG_EXT_CFG_CORE_0) = 0xac0c1e09; REG32(MEMORY_TAG_EXT_CFG_CORE_1) = 0xac0c1e09; REG32(MEMORY_TAG_EXT_CFG_CORE_2) = 0xac0c1e09; ... </pre>
2. Assembly Code for RISC-V
<pre> ... lui s3,0x20010 lui a5,0xac0c2 addi a5,a5,-503 # ac0c1e09 <_stack+0xac0bc091> sw a5,-976(s3) # 2000fc30 <_stack+0x20009eb8> lui s2,0x20011 sw a5,-976(s2) # 20110c30 <_stack+0x2000aeb8> lui s1,0x20012 sw a5,-976(s1) # 20111c30 <_stack+0x2000beb8> ... </pre>

- **Address and Offset Definition:** The `#define` preprocessor directives establish the `BASE_ADDR_SFR_MAP` as the logical starting point for hardware control. Specifically, the `0x0c30` offset points to the `MEMORY_TAG_EXT_CFG` registers, which serves as a core-specific configuration unit for MTE.

- Atomic Configuration of Security Parameters: The statement `REG32(...)=0xac0c1e09` is more than a simple assignment; it is a critical initialization step that commits multiple configuration parameters to the hardware in a single atomic memory write operation.

Appendix A.2. Bit-Field Encoding of a Representative Control Register Value

The configuration value `0xac0c1e09` is a combination of several parameters used to illustrate how multiple control fields are encoded into a single register value:

- Global MTE Enable (Bit [31] : 0x1): Setting the bit to 1 activates the hardware tag comparison engine within the respective processor core. If this bit is not set, all memory access tag checks are bypassed regardless of other settings.
- Tag Check Guard Region Configuration (Bit [30:24]: 0x2c): The value `0x2c` defines the granularity and range for tag validation during memory access. This communicates the physical size of the “Guard Region” to the hardware.
- Hardware Tag Generation Algorithm (Bit [23:16]: 0x0c): The `0x0c` identifier selects the hardware-supported algorithm for memory tag generation, such as sequential or random-based methods.
- Tag Mismatch Interrupt and Exception Policy (Bit [15:8]: 0x1e): The value `0x1e` instructs the hardware to trigger an immediate interrupt or security exception upon tag mismatch detection to prevent data leakage.
- Per-Core Security Domain ID (Bit [7:0]: 0x09): The `0x09` serves as a unique Security Domain ID to facilitate multi-core resource isolation and prevent unauthorized tag access across domains.

Appendix A.3. RISC-V Assembly Translation of High-Level Configuration Code

The RISC-V ISA enforces a fixed instruction length of 32 bits, which creates a fundamental challenge for the management of full-width addresses or constants. A single instruction must allocate bits for the opcode and register identifiers, so it lacks sufficient space to store a full 32-bit immediate value. To address such a physical limitation, the compiler generates a multi-step sequence to reconstruct large values in the register file.

Appendix A.3.1. Immediate Value Generation Using LUI/ADDI Sequences

Because all RISC-V instructions are exactly 32 bits long, a single instruction cannot encapsulate a full 32-bit immediate value. Consequently, a two-step technique is used to load constants into registers:

- `lui a5, 0xac0c2`: The Load Upper Immediate (LUI) instruction loads the upper 20 bits of the `a5` register with `0xac0c2` and clears the lower 12 bits to zero, creating an intermediate value of `0xac0c2000`.
- `addi a5, a5, -503`: An Add Immediate (ADDI) instruction adds the 12-bit signed immediate `-503` (`0xFFFFFE09`) to the value in `a5`. This operation utilizes sign extension, resulting in the final desired security policy value of `0xac0c1e09` within the register. This method is highly efficient as it constructs the 32-bit value entirely within the instruction pipeline without requiring a separate data memory access.

Appendix A.3.2. Address Management and Storage Strategy

The same constant-loading technique is applied to designate the hardware address:

- `lui s3, 0x20010`: This loads the base address of the SFR map (`0x20010000`) on the system bus into the `s3` register, setting the base pointer.

- `sw a5, -976(s3)`: The Store Word (SW) instruction writes the prepared MTE configuration value (a5) to the physical address calculated as `s3 - 976`. The resulting address, `0x2000fc30`, corresponds exactly to the Core 0 MTE configuration register in the hardware specification³⁰.

The compiler retains the security configuration value within the a5 register to optimize execution, so the system avoids redundant immediate generation cycles for subsequent configurations. The engine updates the base address registers for the memory-mapped special function registers to target additional units. Specifically, the compiler issues instructions such as `lui s2, 0x20011` and `lui s1, 0x20012` to establish new base pointers while it utilizes the same store offset of -976 to initialize the target registers at `0x20010c30` and `0x20011c30`. This strategy reduces the total instruction count and ensures that binary modifications by the AIPR framework remain consistent across all designated hardware addresses.

As discussed above, the integrated assembly structure reduces code size and enhances execution performance through combined base address settings and fixed-offset access. Understanding these mechanisms is essential for analyzing how high-level configuration code is lowered into instruction sequences and how such sequences can later be transformed through binary rewriting without altering program semantics.

Appendix A.4. Impact of Arithmetic Sign-Extension Constraints on Binary Parameter Modification

To demonstrate the practical application of binary rewriting, a specific case of modifying system parameters is examined. This process illustrates the transition from high-level policy changes to low-level assembly transformations. The conversion highlights the primary challenge in binary rewriting, as the engine ensures that a modification propagates correctly across dependent instruction sequences. An MTE configuration is used solely as a representative example to demonstrate the complexity of recalculating split immediates and maintaining semantic consistency during the rewriting process.

Suppose the security requirements are elevated to require a Synchronous Error Reporting mode instead of a simple interrupt. The new configuration value is defined as `0xbc0c3e01`. The modified bit-field breakdown is as follows:

- Bit [30:24]: `0x3c`: Guard Region Size increased for stricter memory boundary checks (Changed from `0x2c` → `0x3c`).
- Bit [15:8]: `0x3e`: Exception Policy updated to Synchronous Fault to halt execution immediately upon mismatch (Changed from `0x1e` → `0x3e`).
- Bit [7:0]: `0x01`: Security Domain ID updated to a restricted kernel-level domain (Changed from `0x09` → `0x01`).

In RISC-V, modifying a 32-bit constant like `0xbc0c3e01` is not a localized change. Because the value is split across an LUI (Load Upper Immediate) and an ADDI (Add Immediate) instruction, a binary rewriter must perform a multi-instruction calculation to maintain semantic integrity.

To load `0xbc0c3e01`, the rewriter must account for the sign-extension behavior of the ADDI instruction. If the lower 12 bits of the target value are $\geq 0x800$, the upper 20 bits must be incremented by 1 to compensate for the signed addition.

- Lower 12 bits: `0xbc0c3e01` → `0xe01`. Since `0xe01` $\geq 0x800$, the value is treated as a negative number in two's complement (`0xe01 - 0x1000 = -511`).
- Upper 20 bits: `0xbc0c3` → `0xbc0c4`. The base value `0xbc0c3` must be incremented to account for the subtraction in the next step: `0xbc0c3 + 1 = 0xbc0c4`.

The resulting modified assembly sequence is as follows:

- Original: `lui a5, 0xac0c2` → New: `lui a5, 0xbc0c4`
- Original: `addi a5, a5, -503` → New: `addi a5, a5, -511`

The binary rewriter identifies the LUI/ADDI pair using data-flow analysis and patches the immediate fields of the opcodes with these new calculated values.

Appendix A.5. Instruction-to-Machine-Code Field Transformation

The transformation of machine code observed in Table A2 represents a physical reconfiguration of bit-fields according to the RISC-V ISA specification. While assembly instructions serve as symbolic abstractions for developers, machine code consists of precise bit patterns that the hardware pipeline interprets directly to execute operations. The AIPR framework facilitates the direct manipulation of these bit patterns to enforce new system policies without source-level intervention.

Table A2. Instruction-Level Mapping Between Original and Patched Machine Code. Gray-shaded cells identify the target instruction sequence for the binary rewriting.

Original Machine Code	Original Assembly	Patched Machine Code	Patched Assembly
200109b7	lui s3,0x20010	200109b7	lui s3,0x20010
ac0c27b7	lui a5,0xac0c2	bc0c47b7	lui a5,0xbc0c4
e0978793	addi a5,a5,−503	e0178793	addi a5,a5,−511
c2f9a823	sw a5,−976(s3)	c2f9a823	sw a5,−976(s3)
20011937	lui s2,0x20011	20011937	lui s2,0x20011
c2f92823	sw a5,−976(s2)	c2f92823	sw a5,−976(s2)
200124b7	lui s1,0x20012	200124b7	lui s1,0x20012
c2f4a823	sw a5,−976(s1)	c2f4a823	sw a5,−976(s1)
6a99	c.lui s5,0x6	6a99	c.lui s5,0x6

The transition from the original machine code to the patched version is a result of targeted immediate field modifications within fixed-length 32-bit instruction words. This process preserves the integrity of the opcode and register fields while updating the architectural parameters.

Appendix A.5.1. LUI Instruction Update

The LUI (Load Upper Immediate) instruction follows the U-type format, which is architecturally designed to handle the upper 20 bits of a 32-bit constant.

- Original Machine Code (0xac0c27b7): This 32-bit word is composed of the opcode (0x37 for LUI), the destination register a5, and the original immediate field 0xac0c2.
- Patched Machine Code (0xbc0c47b7): To implement the elevated security policy (0xbc0c3e01), the AIPR framework targets the immediate bit-field (bits 31:12). By calculating the new value 0xbc0c3 and applying the sign-extension compensation to reach 0xbc0c4, the framework physically rewrites the upper bits of the instruction. The opcode and register fields remain static, ensuring the instruction still targets a5 but with the updated policy bits.

Appendix A.5.2. ADDI Instruction Update

The ADDI (Add Immediate) instruction utilizes the I-type format, where a 12-bit signed immediate is added to a source register.

- Original Machine Code (0xe0978793): The bits 31:20 contain the original 12-bit immediate value of −503 (0xe09 in hex).
- Patched Machine Code (0xe0178793): The AIPR engine replaces this specific 12-bit immediate field with the new calculated value of −511 (0xe01 in hex). Because this modification is strictly localized to the immediate bit-field, the internal data flow—adding a constant to a5 and storing the result back to a5—is maintained without side effects on the surrounding hardware logic.

Appendix A.6. Binary Rewriting via Hexadecimal Stream Manipulation

After establishing the instruction-level transformations at the machine code level, the analysis now shifts to the actual executable representation, where modifications must be applied directly to the hexadecimal data stream. At this level, symbolic instruction boundaries no longer exist explicitly. Instead, instructions are embedded within a continuous byte stream, and any rewriting operation must precisely identify the correspondence between instruction semantics and their physical byte locations as shown in Table A3.

While the AIPR framework demonstrates high functional reliability, several architectural constraints must be addressed to ensure the generalizability of binary rewriting across diverse RISC-V implementations. A fundamental consideration involves the little-endian memory layout of the RISC-V architecture. The system stores the least significant byte at the lowest address. Consequently, the framework modifies these bytes in reversed order to prevent execution failure. Furthermore, instruction length variability introduces complexity because the compressed instruction extension (RVC) allows 16-bit instructions to coexist with standard 32-bit instructions. The presence of RVC disrupts 4-byte alignments. The engine detects instruction boundaries and respects half-word alignment to avoid the accidental corruption of adjacent code. Table A4 summarizes the technical mapping between machine code and the final hexadecimal modifications within the executable image after the completion of instruction-level analysis and validation.

Table A3. Little-Endian Hexadecimal Byte Stream of the Executable showing Instruction Alignment and Target Offsets. Gray-shaded cells identify the target instruction sequence for the binary rewriting.

Address Offset		Hexadecimal Data (32-Bit Words)			
@00000c04	23a20790	23a40790	b7090120	b7270cac	
@00000c08	938797e0	23a8f9c2	37190120	2328f9c2	
@00000c0c	b7240120	23a8f4c2	996a9387	fa4023ae	
@00000c10	f9c0232e	f9c023ae	f4c02945	eff0dfe0	
@00000c14	93873a40	23aef9c0	232ef9c0	23aef4c0	

Table A4. Unified Mapping of Hexadecimal Patches and Architectural Constraints.

Address	Instruction	Original Hex (LE)	Patched Hex (LE)	Key Constraint
@00000c04	LUI a5	b7270cac	b7470cbc	Sign-extension
@00000c08	ADDI a5	938797e0	938717e0	12-bit immediate limit
@00000c0c	C.LUI s5	996a	996a	Half-word alignment

Appendix B. Implementation Direction for MTE in V-FRONT RTL

To implement Memory Tag Extension (MTE) functionality within the V-FRONT microarchitecture, targeted modifications are applied to the Control and Status Register (CSR) unit, the Memory (MEM) stage logic, and the exception handling mechanism. The purpose of this implementation is not to provide a complete, production-grade realization of MTE, but rather to introduce a controlled yet non-trivial architectural feature that enables realistic instruction-level transformation and validation scenarios.

V-FRONT is a simplified RV32I-compliant processor, intentionally selected to minimize confounding effects from vendor-specific optimizations or advanced microarchitectural features. Within this controlled setting, the proof-of-concept MTE implementation emulates essential architectural behaviors—such as tag comparison, policy enforcement, and exception triggering—at a level sufficient to rigorously verify the functional robustness and performance efficiency of the AIPR framework.

This RTL-level augmentation provides a representative validation environment in which executable-level binary rewriting can be evaluated under realistic architectural

constraints, without overstating generality across all RISC-V implementations. Validation on additional cores, ISA extensions, and more diverse binary layouts is intentionally left as future work to preserve experimental clarity and scope. Detailed hardware implementation aspects remain beyond the scope of this paper. The following sections outline the conceptual RTL structure.

The core logic for MTE resides in the MEM stage, where the physical address tag is compared with the stored tag metadata.

Table A5. RTL Implementation Direction for MTE Functionality in V-FRONT Processor.

1. MTE Configuration Register (CSR) Logic
<pre>// Dedicated CSR to store security parameters reg [31:0] mte_config_reg; wire global_mte_en = mte_config_reg[31]; // Bit [31]: Global Enable wire [6:0] guard_sz = mte_config_reg[30:24]; // Bit [30:24]: Guard Region Size wire [7:0] fault_pol = mte_config_reg[15:8]; // Bit [15:8]: Exception Policy wire [7:0] domain_id = mte_config_reg[7:0]; // Bit [7:0]: Security Domain ID</pre>
2. Tag Comparison and Exception Generation (MEM Stage)
<pre>// Hardware-level tag validation during memory access wire [3:0] stored_tag; // Metadata retrieved from Tag RAM wire [3:0] provided_tag = mem_addr_in[31:28]; // Top 4 bits used as Tag // Mismatch detection logic wire tag_mismatch = (provided_tag != stored_tag) && global_mte_en; // Exception triggering based on Synchronous Fault policy assign mte_fault_signal = (tag_mismatch && (fault_pol == 8'h3E)) ? 1'b1 : 1'b0;</pre>
3. Pipeline Flush and Exception Handling
<pre>// Integrated trap logic to halt execution upon mismatch always @(*) begin if (mte_fault_signal) begin pipeline_flush = 1'b1; // Invalidate following instructions exception_vector = 32'h0000_0100; // Jump to security handler end end</pre>

References

1. Akram, A.; Sawalha, L. A survey of computer architecture simulation techniques and tools. *IEEE Access* **2019**, *7*, 78120–78145.
2. Neelu Kumari, K.S.; Murali, L.; Vijayabaskar, S.; Gopalakrishnan, R. A Reconfigured Architecture of Mathematical Morphology Using Fuzzy Logic Controller for ECG QRS Detection. *J. Electr. Eng. Technol.* **2025**, *20*, 1789–1802.
3. Vieira, J.; Roma, N.; Falcao, G.; Tomás, P. gem5-accel: A pre-rtl simulation toolchain for accelerator architecture validation. *IEEE Comput. Archit. Lett.* **2023**, *23*, 1–4.
4. Karandikar, S.; Mao, H.; Kim, D.; Biancolin, D.; Amid, A.; Lee, D.; Pemberton, N.; Amaro, E.; Schmidt, C.; Chopra, A.; et al. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, USA, 2–6 June 2018; pp. 29–42.
5. Jiang, F.; Maeda, R.K.; Feng, J.; Chen, S.; Chen, L.; Li, X.; Xu, J. Fast and accurate statistical simulation of shared-memory applications on multicore systems. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *33*, 2455–2469.

6. Purraji, M.; Zamiri, E.; Sanchez, A.; de Castro, A. Rapid Prototyping for Design and Test of FPGA-Based Model Predictive Controllers for Power Converters. *J. Electr. Eng. Technol.* **2025**, *in press*.
7. Perkel, J.M. Democratic databases: Science on GitHub. *Nature* **2016**, *538*, 127–128.
8. Lowndes, J.S.S.; Best, B.D.; Scarborough, C.; Afflerbach, J.C.; Frazier, M.R.; O'Hara, C.C.; Jiang, N.; Halpern, B.S. Our path to better science in less time using open data science tools. *Nat. Ecol. Evol.* **2017**, *1*, 0160.
9. Collberg, C.; Proebsting, T.A. Repeatability in computer systems research. *Commun. ACM* **2016**, *59*, 62–69.
10. Sharifi, S.; Reuel, N.; Kallmyer, N.; Sun, E.; Landry, M.P.; Mahmoudi, M. The issue of reliability and repeatability of analytical measurement in industrial and academic nanomedicine. *ACS Nano* **2022**, *17*, 4–11.
11. Gundersen, O.E.; Kjensmo, S. State of the art: Reproducibility in artificial intelligence. In Proceedings of the AAAI Conference on Artificial Intelligence, New Orleans, LA, USA, 2–7 February 2018; pp. 1644–1651.
12. Salkhordeh, R.; Brinkmann, A. On the reproducibility of computer architecture research in the era of open source. In Proceedings of the International Conference on Performance Engineering, Virtual, 19–23 April 2021.
13. Konersmann, M.; Kaplan, A.; Kuhn, T.; Heinrich, R.; Koziol, A.; Reussner, R.; Jürjens, J.; al-Doori, M.; Boltz, N.; Ehl, M.; et al. Evaluation methods and replicability of software architecture research objects. In Proceedings of the 19th IEEE International Conference on Software Architecture (ICSA), Honolulu, HI, USA, 12–15 March 2022; pp. 157–168.
14. Goth, F.; Thiele, J.P.; Project, T.T. Foundational competencies and specializations of a research software engineer. *Comput. Sci. Eng.* **2025**, *27*, 27–34.
15. Balas, R.; Benini, L. RISC-V for real-time MCUs: Software optimization and microarchitectural gap analysis. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 1–5 February 2021; pp. 874–877.
16. Duck, G.J.; Gao, X.; Roychoudhury, A. Binary rewriting without control flow recovery. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), London, UK, 15–19 June 2020; pp. 151–163.
17. Wenzl, M.; Merzdovnik, G.; Ullrich, J.; Weippl, E. From hack to elaborate technique—A survey on binary rewriting. *ACM Comput. Surv.* **2019**, *52*, 1–37.
18. Park, J.; Yun, I.; Ryu, S. Bridging the gap between real-world and formal binary lifting through filtered simulation. *Proc. ACM Program. Lang.* **2025**, *9*, 898–926.
19. Scott, R.G.; Boston, B.; Davis, B.; Diatchki, I.; Dodds, M.; Hendrix, J.; Matichuk, D.; Quick, K.; Ravitch, T.; Robert, V.; et al. Macaw: A machine code toolbox for the busy binary analyst. *arXiv* **2024**, arXiv:2407.06375.
20. Mezger, B.W.; Santos, D.A.; Dilillo, L.; Zeferino, C.A.; Melo, D.R. A survey of the RISC-V architecture software support. *IEEE Access* **2022**, *10*, 51394–51411.
21. Hassan, Q.F.; Sagahyoon, A. RISC-V: A comprehensive overview of an emerging ISA for the AI-IoT era. *Adv. Internet Things* **2025**, *in press*.
22. Boubakri, M.; Zouari, B. GATOR-V: Accelerating the RISC-V confidential computing ecosystem with a production-grade TEE. *IEEE Access* **2025**, *13*, 210892–210916.
23. Barker, M.; Chue Hong, N.P.; Katz, D.S.; Lamprecht, A.L.; Martinez-Ortiz, C.; Psomopoulos, F.; Harrow, J.; Castro, L.J.; Gruenpeter, M.; Martinez, P.A.; et al. Introducing the FAIR principles for research software. *Sci. Data* **2022**, *9*, 622.
24. Waterman, A.; Asanović, K. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*; RISC-V Foundation: San Francisco, CA, USA, 2019.
25. Patterson, D.; Waterman, A. *The RISC-V Reader: An Open Architecture Atlas*; Strawberry Canyon: Berkeley, CA, USA, 2017.
26. Dikmen, K. V-FRONT: A Five-Stage 32-bit RISC-V Processor in Verilog. GitHub. Available online: <https://github.com/kagandikmen/V-FRONT> (accessed on 12 January 2026).
27. Zhang, L.; Yang, X.; Cheng, X.; Cheng, W.; Lin, Y. Few-Shot Image Classification Algorithm Based on Global–Local Feature Fusion. *AI* **2025**, *6*, 265. <https://doi.org/10.3390/ai6100265>

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.