electronics

MDPI

Article

CFIEE: An Open-Source Critical Metadata Extraction Tool for RISC-V Hardware-Based CFI Schemes

Wenxin Li, Weike Wang * and Senyang Li

College of Electronic and Information Engineering, Shandong University of Science and Technology, Qingdao 266590, China; wenxin_li@sdust.edu.cn (W.L.); lisenyang@sdust.edu.cn (S.L.)
* Correspondence: wangweike@sdust.edu.cn

Abstract: Control flow critical metadata play a key role in hardware-based control flow integrity (CFI) mechanisms that effectively monitor and secure program control flow based on pre-extracted metadata. The existing control flow analysis tools exhibit some deficiencies, including inadequate compatibility with the RISC-V architecture, a steep learning curve, limited automation capabilities, and restricted data output formats. CFIEE is an open-source tool with a graphical interface for the automated extraction of control flow critical metadata. The tool possesses the capability to analyze RISC-V binary executables, transforming the binary into an intermediate representation (IR) in the form of the disassembled code, and extracting the critical metadata required for studying hardware-based CFI mechanism through a designed control flow transfer relationship analysis algorithm. The extracted metadata include program basic blocks and their corresponding hash values, control flow graphs, function call relationships, distribution of forward transfer instructions, etc. We selected 15 embedded system programs with processor adaptation for functional verification. The results demonstrate the CFIEE's capability to automatically analyze programs within the supported RISC-V instruction set and generate comprehensive and precise metadata files. This tool can significantly enhance the efficiency of control flow metadata extraction and furnish configurable metadata for the hardware-based security mechanisms.

Keywords: RISC-V; control flow integrity; basic block; control flow graph

Citation: Li, W.; Wang, W.; Li, S. CFIEE: An Open-Source Critical Metadata Extraction Tool for RISC-V Hardware-Based CFI Schemes. *Electronics* **2024**, *13*, 1681. https://doi.org/10.3390/electronics13091681

Academic Editor: Luis Gomes

Received: 2 April 2024
Revised: 19 April 2024
Accepted: 26 April 2024
Published: 26 April 2024

1. Introduction

The RISC-V architecture has gained considerable attention in recent years as an open and extensible instruction set architecture (ISA). Known for its modular design and support for customized instructions, RISC-V has become a popular choice for various applications, including embedded systems, Internet of Things (IoT) devices, and high-performance computing. However, with the increasing adoption of RISC-V devices, addressing the architecture's potential security vulnerabilities, particularly in terms of control flow security, has emerged as an urgent concern. Control flow hijacking attacks encompass various techniques such as buffer overflow [1] exploits and return-oriented programming [2]. These techniques allow attackers to manipulate a program's control flow by corrupting or overwriting memory locations that store critical information about function calls or returns. Control Flow Integrity (CFI) [3] mechanisms play a crucial role in modern software security by safeguarding against control-flow hijacking attacks. These mechanisms rely on precise control flow data analysis to ensure the integrity of a program's execution path [4]. The extraction and analysis of control flow metadata facilitate the detection of anomalies, identification of control flow hijacking attempts, and development of effective countermeasures. Techniques such as control flow graphs, basic block identification, and loop detection provide insights into the execution path of a program, aiding in the identification of potential security vulnerabilities. Furthermore, control flow information can be utilized for runtime monitoring and intrusion detection. By comparing the actual control flow with

the expected behavior, deviations and anomalies can be promptly detected, enabling timely responses and the mitigation of security incidents.

Control flow analysis tools are a relatively mature research field. In the past decades, many different control flow analysis methods and tools have been put forward by academia and industry for static analysis [5,6], dynamic analysis [7,8], symbol execution [9,10] and so on. These tools play an important role in identifying the control flow structure of programs, detecting vulnerabilities and optimizing codes. However, the majority of existing control flow analysis tools are primarily tailored for mature architectures like X86 and ARM and thus may not seamlessly integrate with the emerging RISC-V architecture. Additionally, these tools present certain usability barriers such as the requirement of a separate program for invoking the analysis tool. The objective of this paper is to enable automated control flow analysis of RISC-V binary files while also providing an intuitive graphical interface that facilitates quick adoption and relevant research by newcomers in this field.

CFIEE (Control Flow Integrity metadata Extraction Engine) is a Python-based static analysis tool specifically designed for RISC-V architecture. The development of CFIEE is based on the T-Head Xuantie E906 RISC-V processor [11] supporting the RV32IMAFC instruction set [12,13]. It utilizes our pre-designed algorithm to closely approximate the program's actual behavior, analyze the control transfer relationships, and extract control flow information. CFIEE can extract critical information, including basic blocks, control transfer instructions, function addresses, control flow graph, and sensitive data crucial for CFI verification. Our contributions are summarized as follows:

- The CFIEE tool is a critical metadata extraction tool for RISC-V hardware-based CFI schemes, providing output data files that serve as valuable references for the design of hardware-based CFI mechanisms.
- We have developed an algorithm for analyzing control transfer relationships based on the execution rules of RISC-V programs. Through static analysis, the algorithm can approximate the actual execution path of the program, providing CFIEE with a comprehensive analysis scope, which in turn provides researchers with comprehensive CFI metadata.
- CFIEE will be released as an open-source project [14], providing unrestricted usage and modification of the software to all individuals under an open-source license.

The paper is structured as follows: Section 2 introduces the control transfer instructions in the RISC-V instruction set, explains the concept of control flow graph, and discusses the working phases of the CFG-based CFI mechanism. In Section 3, a detailed explanation is given regarding the software architecture, internal workflow, functions of different components, and output files of CFIEE. Section 4 presents an application scenario where CFIEE offers data support for hardware-based CFI mechanisms. In Section 5, a comparison between CFIEE and tools with similar functionalities is made along with showcasing analysis results of CFIEE on test programs. Finally, conclusions are presented in Section 6.

2. Background and Related Works

2.1. Control Transfer Instructions in RISC-V ISA

The RISC-V Instruction Set Architecture (ISA) has emerged as a significant force in computer architecture and microprocessor design. It is an open standard instruction set architecture that has gained widespread attention and adoption in academia and industry due to its versatility, extensibility, and flexibility. The RISC-V ISA adheres to the principles of Reduced Instruction Set Computing (RISC), emphasizing simplicity and efficiency. This architectural elegance is evident in its streamlined instruction set, which allows instructions to execute in a single clock cycle, optimizing performance and energy efficiency [15]. One of the key features of RISC-V is its modularity. The ISA is structured around a base integer instruction set, providing a foundation for various application-specific extensions. This modular design enables tailored customization by incorporating specialized instructions to address specific computational needs while ensuring compatibility with the core ISA. Additionally, the RISC-V ISA supports both 32-bit and 64-bit address spaces [16],

accommodating a wide range of computing platforms and applications. This adaptability makes RISC-V suitable for deployment in resource-constrained embedded systems and high-performance computing environments.

Table 1 showcases the conditional branch instructions present in the RV32IMAFC instruction set [12,13], excluding pseudo-instructions. When a conditional branch instruction is executed, it involves comparing the values of two source registers (rs1 and rs2), and based on the result, the branch may or may not be taken. This decision-making process underpins the core of conditional branch instructions, allowing programs to take different execution paths based on logical conditions.

Table 1. Conditional branch instructions in RV32IMAFC.

Name	Mnemonic
branch equal	beq
branch not equal	bne
branch less than	blt
branch greater than or equal	bge
branch less than unsigned	bltu
branch greater than or equal unsigned	bgeu

Table 2 illustrates the unconditional jump instructions within the RV32IMAFC instruction set, excluding pseudo-instructions. The "jal" instruction, an essential member of this category of instructions, is an abbreviation for "jump and link". Upon execution, it unconditionally jumps to a specific section of the program while simultaneously storing the return address in the "x1" register, which is commonly referred to as the "ra" register. In contrast, the "j" instruction represents another form of unconditional jump instruction within the RV32IMAFC instruction set. Like "jal," it unconditionally diverts program execution to a designated location. However, unlike "jal," the "j" instruction does not undertake the task of preserving the return address. The "jalr" instruction represents an additional aspect of the RV32IMAFC instruction set, embodying the concept of indirect jumps where the target address is not explicitly specified in the disassembly code but derived from the contents of rs1 register. This flexibility in specifying jump targets lends itself to various programming scenarios where dynamic or indirect addressing is required.

Table 2. Unconditional jump instructions in RV32IMAFC.

Name	Mnemonic
jump and link	jal
jump	j
jump and link register	jalr

2.2. Control Flow Graph

Control Flow Graph (CFG) is a graphical structure utilized for representing the program's control flow, which is typically in the form of a directed graph [17]. The CFG nodes are commonly referred to as basic blocks, which represent uninterrupted code units within the program. Different basic blocks are usually connected by control flow edges, which are directed edges that connect different basic blocks in the CFG. These edges represent the jump or branch relationships during program execution, signifying that upon completion of one basic block's execution, control flow will be transferred to another basic block.

The control flow edges can be categorized into forward and backward edges [18]. A forward edge represents the normal direction of control flow in a program, that is, a directed edge from one basic block to another. This type of edge represents the program's control flow transfer along the normal execution path. For instance, forward edges arise when program execution proceeds sequentially to the next basic block or when the true path of a conditional branch is executed. Backward edges are utilized to represent loops or conditional branches in a program, enabling the program to backtrack from one basic block

to a previously executed basic block. These edges reflect the non-linear control flow of the program. For instance, within a loop structure, a backward edge occurs when an iteration is completed and the program returns to the beginning of the loop, facilitating multiple executions of code within its body.

2.3. Phases of CFG-Based CFI Mechanisms

Most CFI mechanisms can be categorized into two distinct phases [19], each contributing to the overarching goal of enhancing program security.

CFG Construction and Analysis: In the first stage, the CFI mechanism needs to obtain the CFG of the program through a specific analysis process. The accuracy and comprehensiveness of the CFG directly influence the effectiveness of the control flow policy. There are three different approaches to construct control flow graphs: static, dynamic and hybrid. Static analysis is a prevalent technique for constructing CFGs. This method involves a meticulous examination of the program, such as the source code and binary executable file [20–22]. Static analysis is often conducted during the program's compilation or preprocessing phase, ensuring that the CFG is established before execution. One of the dynamic CFG reconstruction methods was proposed by Yount et al. [23]. Dynamic analysis takes a different approach by constructing the CFG during program execution [24]. This real-time approach allows the mechanism to adapt to the program's actual behavior, ensuring that the CFG accurately reflects runtime conditions. While dynamic analysis can provide a precise representation of the program's control flow, it may introduce some overhead due to the need for continuous monitoring during execution. Nonetheless, it is a valuable technique for scenarios where the control flow structure may change dynamically. V.H. Sahin proposed Turna, which is a tool for building control flow graphs using a hybrid approach [25]. Hybrid approaches combines static and dynamic analysis methods, static analysis provides a framework for the initial control flow graph, and then dynamic analysis is used to refine the graph or verify the control flow. For example, a hybrid approach may use static analysis to establish an initial CFG and then dynamically refine it during program execution to account for runtime variations. Once the CFG is established, the mechanism defines the permissible control flow transfer targets based on CFG.

Runtime Control Flow Verification: The second phase of the CFI mechanism, which takes place during runtime when the program is being executed, plays a crucial role in ensuring the security and integrity of the running program. During this phase, the CFI mechanism continuously monitors the control flow transfers within the running program [26]. By continuously monitoring these control flow transfers, the CFI mechanism aims to verify whether they adhere to the predetermined Control Flow Graph (CFG) constructed in the first phase [27]. The CFG serves as a blueprint for legitimate control flow paths within the program. Any attempt to transfer control outside of this predefined range raises suspicion and triggers a security response. When the CFI mechanism detects an unauthorized control flow that lacks verification and deviates from the expected path, it promptly initiates appropriate security measures to mitigate potential threats. These security responses can vary depending on system configurations and requirements but may involve terminating or suspending execution of the program. In addition to halting execution, performing exception handling procedures becomes essential when dealing with unauthorized control flow transfers. Exception handling allows for graceful recovery from unexpected events or errors encountered during runtime. By employing proper exception handling techniques, developers can ensure that any abnormal termination caused by unauthorized control flow transfers does not result in data corruption or system instability.

3. Technical Specifications

CFIEE is a critical metadata extraction tool for RISC-V hardware-based CFI mechanisms implemented in Python. It is compatible with any computer system that supports Python 3. To utilize its disassembly functionality, the tool requires the installation of the riscv32/64-unknown-elf toolchain on the user's computer system.

3.1. Overview of CFIEE Architecture

The CFIEE architecture, as illustrated in Figure 1, offers a comprehensive depiction of its functionality. This tool accepts either an RISC-V executable or disassembled file as input, which subsequently undergoes processing through three distinct processes within the CFIEE framework. Ultimately, it generates metadata files pertaining to CFI.

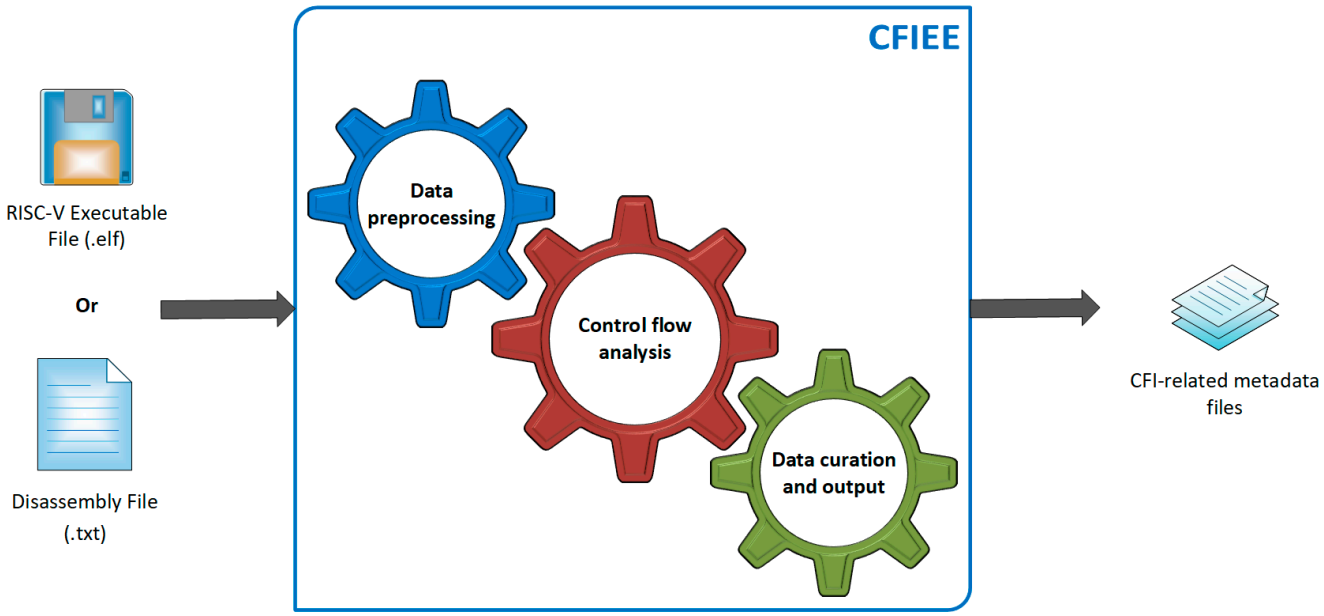


Figure 1. Overview of CFIEE architecture.

3.1.1. Input Files

In scenarios such as reverse engineering and malware analysis, it is frequently encountered to have only binary files without access to the corresponding source code. The behavior of a program can be better understood by analyzing compiled binaries and obtaining actual execution path information. Considering this aspect, CFIEE utilizes binary files as the foundation for analysis. CFIEE is capable of accepting RISC-V executables as input. Specifically, the tool can analyze ELF files generated by compiling under the RV32IMAFC instruction set. CFIEE ensures proper analysis when the program utilizes an instruction set within this range.

Additionally, CFIEE can process disassembled files in TXT format if provided by the user. In such cases, users can pre-disassemble the executable file using the RISC-V toolchain and save the resulting disassembly as a .txt file. This flexibility in input format widens the tool's applicability, catering to varying user preferences and simplifying the analysis process.

3.1.2. Internal Processes

The internal process of CFIEE is illustrated in Figure 1, encompassing three fundamental components: data preprocessing, control flow analysis, and data curation and output. The "data preprocessing" phase is dedicated to formatting the contents of the disassembly file to adhere to CFIEE's processing format. This crucial step aims to eliminate any extraneous content that may result from specific compilation options during program compilation. Preprocessing ensures the extraction of disassembly instructions, enabling smooth subsequent processing.

The core of CFIEE lies in the "control flow analysis" stage. Starting with the initialization function of the program, CFIEE examines and analyzes the control flow of the program. The analysis process includes extracting potential executable functions, decrypting the control flow transfer relationship in these functions, and identifying each basic block.

The "data curation and output" phase primarily concentrates on consolidating the valuable information acquired during the preceding stages and presenting it in either textual or graphical formats. These organized data are then outputted into appropriate files, facilitating further analysis.

3.1.3. CFI-Related Metadata Files

Table 3 showcases the output files of CFIEE. As of the current version, the tool generates eight output files, including three text files associated with basic blocks, three files regarding control transfer instructions, a control flow diagram represented as a vector diagram, and a function call diagram. Notably, a binary file in the bin format is established, containing all forward transfers' addresses. Each line consists of a 32-bit binary number, where the initial 16 bits represent the binary address of the jump instruction, and the final 16 bits delineate the target address of the jump instruction. These documents can provide data reference for CFI scheme.

Table 3. Output files of CFIEE.

Filename	File Type	Introduction
xxx_basic_block	.txt	Basic blocks' information
xxx_bin_basic_block_info	.txt	Blocks' info in binary form
xxx_hex_basic_block_info	.txt	Blocks' info in hexadecimal form
xxx_forward_transfers	.txt	All forward transfer instructions and target instructions
xxx_control_transfer	.bin	Metadata related to forward transfers
xxx_CFG	.svg	Program-wide control flow graph
xxx_forward_transfers_per_function	.svg	Show the number of transfer instructions within each function
xxx_function_call_relationship	.svg	Demonstrate the program's function call relationships

3.2. Workflow of CFIEE

This section introduces the workflow of CFIEE. Figure 2 showcases the step-by-step workflow process.

It begins with an initial input or data collection stage, followed by multiple analysis and processing steps, and concludes with generating desired outputs or results. If the input file is an RISC-V executable file, CFIEE will invoke the RISC-V toolchain in the data preprocessing module to disassemble it, generating a disassembly code file in TXT format. Depending on the file format, CFIEE will then proceed to extract it for instruction recognition or retain it in its original format.

The preprocessed disassembly code will be forwarded to the analysis module for control flow analysis. CFIEE initially extracts function-related data from the disassembly code, identifying all potentially executable functions within the program. Subsequently, it analyzes the control flow transition relationships, which is followed by partitioning the disassembly code into basic blocks within the range of executable functions.

In the data collation and output module, CFIEE computes the hash value of each basic block based on the basic block instructions using the hash algorithm specified by the user and consolidates the basic block information. Additionally, the module handles the sorting and output of control transfer instructions. It gathers control transfer instructions within each function, identifies their corresponding target instructions, and pairs them accordingly. This module also organizes information pertaining to output functions, including the count of forward control transfer instructions within each function and the program's function call relationships.

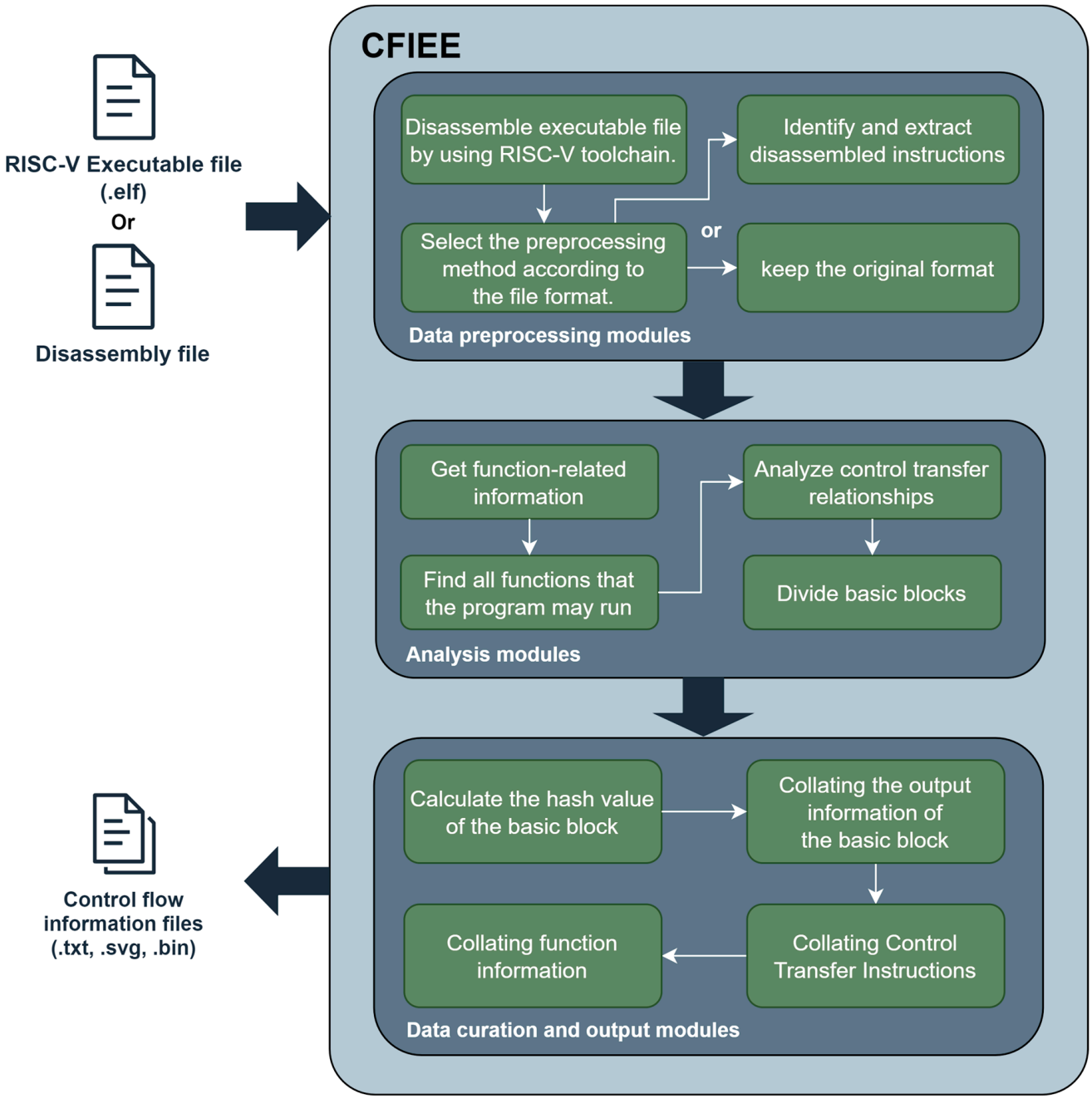


Figure 2. Workflow of CFIEE.

3.3. Functions of CFIEE

The detailed presentation of CFIEE's functions is illustrated in Figure 3, showcasing the key functionalities embedded within the tool's source code, encompassing data manipulation, statistical analysis, and data visualization.

3.3.1. Data Preprocessing

During the data preprocessing stage, CFIEE offers the disassembly functionality for ELF files, automatically selecting the disassembled file once the disassembly process is finalized. In situations where the input files are already in a disassembled format, CFIEE takes an adaptive approach to tailor its processing recommendations based on the specific format of these files. This intelligent adaptation ensures that optimal preprocessing decisions are presented to the user regardless of whether they are working with raw

binary or pre-disassembled files. The execution of the data preprocessing process is jointly managed by the "file_preprocess.py" and "CFIEE.py" scripts.

As shown in Figure 3, the process consists of three main functions. The function "judge_file_type()" is housed in "CFIEE.py" and aims to determine the need for additional processing of the current disassembly file based on pre-established rules. This function will provide a tag value to subsequent related functions based on the file format. The functions "extract_disassemble_introduction()" and "rewrite_objdump_file()" are situated in the script file "file_preprocess.py". Their respective responsibilities involve extracting the necessary instructions from the disassembly file and reconstructing it. The restructured files reside in the same directory as the source files. This systematical approach ensures the efficient and accurate extraction of the required instructions while eliminating redundant information to enable subsequent analysis within the CFIEE framework.

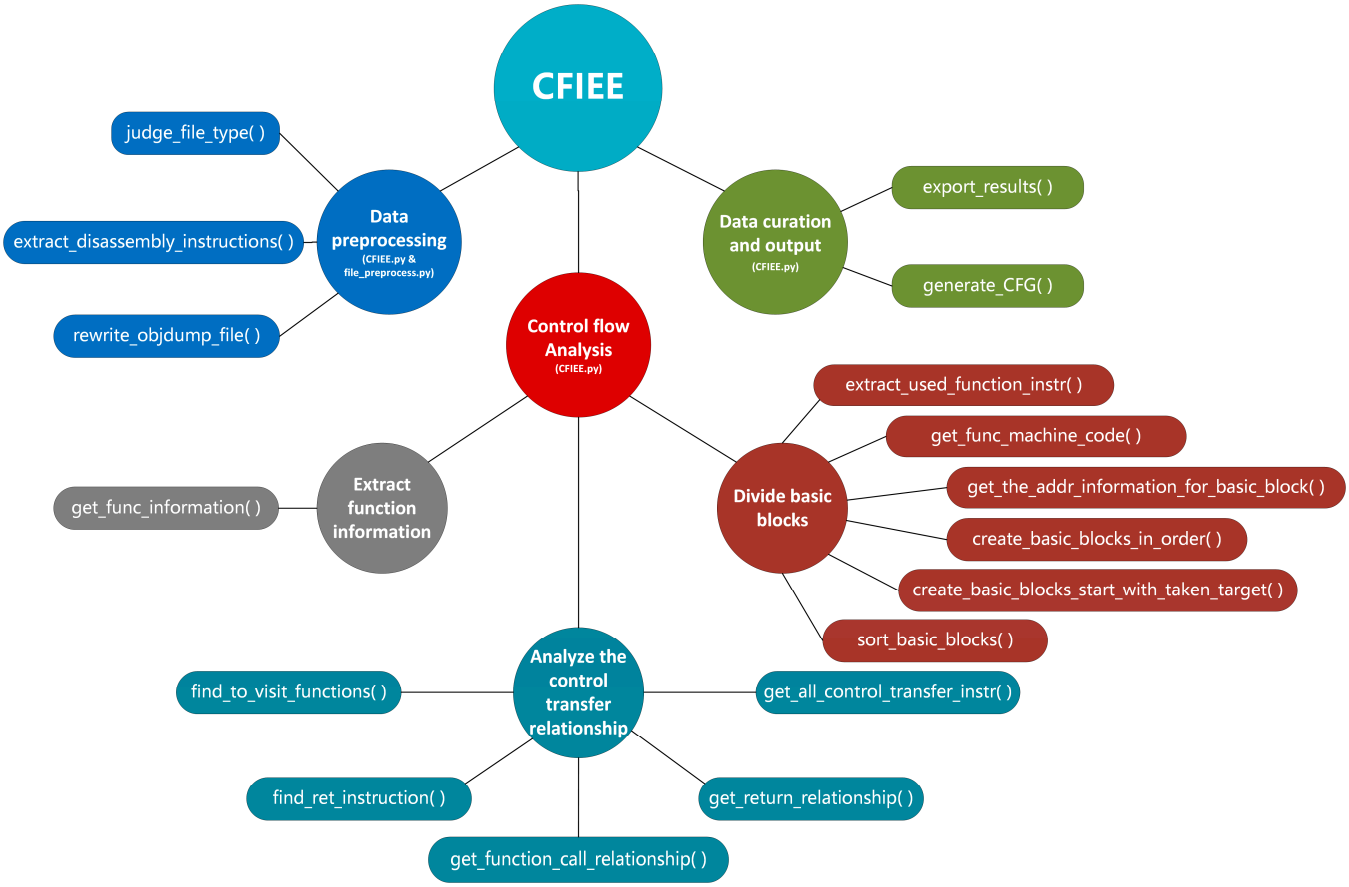


Figure 3. CFIEE's function composition.

3.3.2. Control Flow Analysis

The overall process is divided into three parts: "extract function information", "analyze control transfer relationship", and "divide basic blocks". To begin with, the analysis modules receive the disassembled file as input. The tool starts by extracting various details from the disassembly file, such as function names, start and end addresses, and instruction locations. This initial extraction provides a foundation for further analysis.

Next, CFIEE employs a recursive search algorithm based on program logic to analyze each function. By scrutinizing transfer instructions within these functions, CFIEE anticipates the target addresses the program will access during execution. If any transfer instructions are found within the specific function under analysis, CFIEE delves into the functions corresponding to those target addresses for further examination. Algorithm 1 shows the specific logic of the algorithm.

Algorithm 1. Find to_visit functions

Input: disassemble_file_info; func_name; function_call_instr; visited_functions
Output: to_visit_functions; visited_functions; function_call_instr

```
1.  if visited_functions_id is None:
        initialize visited_functions set
2.  Initialize function_call_instr
3.  Read disassemble file and store the lines in the variable lines
4.  Get the function address range for the current function
5.  Search for called functions in the function range:
      for each line in function_instr[func_name]:
        if instruction is 'jal' or 'j':
          jump_target <- Get jump target operand
        if jump_target is outside func_addr_range:
          Append line to call_instrs
        else if jump_target is within any function's address range:
          Add corresponding function name to to_visit_functions
          called_func_name <- func_name
        else if instruction is branch instruction:
          jump_target <- Get jump target operand
          if this is the last instruction in current function:
            Add next function's name to to_visit_functions
            called_func_name <- func_name
          if jump_target is within any function's address range:
            Add corresponding function name to to_visit_functions
            called_func_name <- func_name
6.  Update function_call_instr dictionary
7.  if no called functions found:
        Add the next sequential function as to visit
8.  Recursively search for called functions in the called functions
9.  Return to_visit_functions, visited_functions_id, visited_functions, and function_call_instr
```

It is based on the transfer instructions within the function. If there are transfer instructions inside the function that the program is currently analyzing, CFIEE will analyze the function where these destination addresses are located. If there are no jump instructions within the current function, CFIEE will mark the next function adjacent to it as a possible function to execute.

After extracting the functions that are likely to be executed, the tool analyzes the control transfer relationships within these functions. It identifies all control transfer instructions and analyzes the target addresses based on the type of transfer instruction. Figure 4 illustrates the analysis logic of CFIEE for function call and return relationships. Prior to analyzing the function call and return relationships, CFIEE has extracted all function address ranges, function call instructions and function return instruction data. When analyzing the function call relationships, whether a function call is generated is determined by the target address of an unconditional jump instruction and the starting address of a specific function. If the condition is true, the call relationship between the current function and the jump target function will be established.

In the process of analyzing the disassembly, a pattern of function calls similar to nested calls caught our attention. For instance, function 1 contains a "jal" instruction that will unconditionally jump to function 2 after saving the return address. At the end of function 2, there is an unconditional jump instruction of type "j". When the program reaches this point, it will jump directly to function 3. Finally, the program executes the return operation at the return instruction of function 3. In response to this scenario, we developed the corresponding analysis logic and incorporated it into the function call relationship. Once the analysis of the function call relationship is completed, it will serve as reference data for

analyzing the function return relationship. The analysis approach for the function return relationship is similar to the previous analysis. CFIEE will analyze the function return relationship and determine the target address based on the function call relationship and the address information of the "ret" instruction.

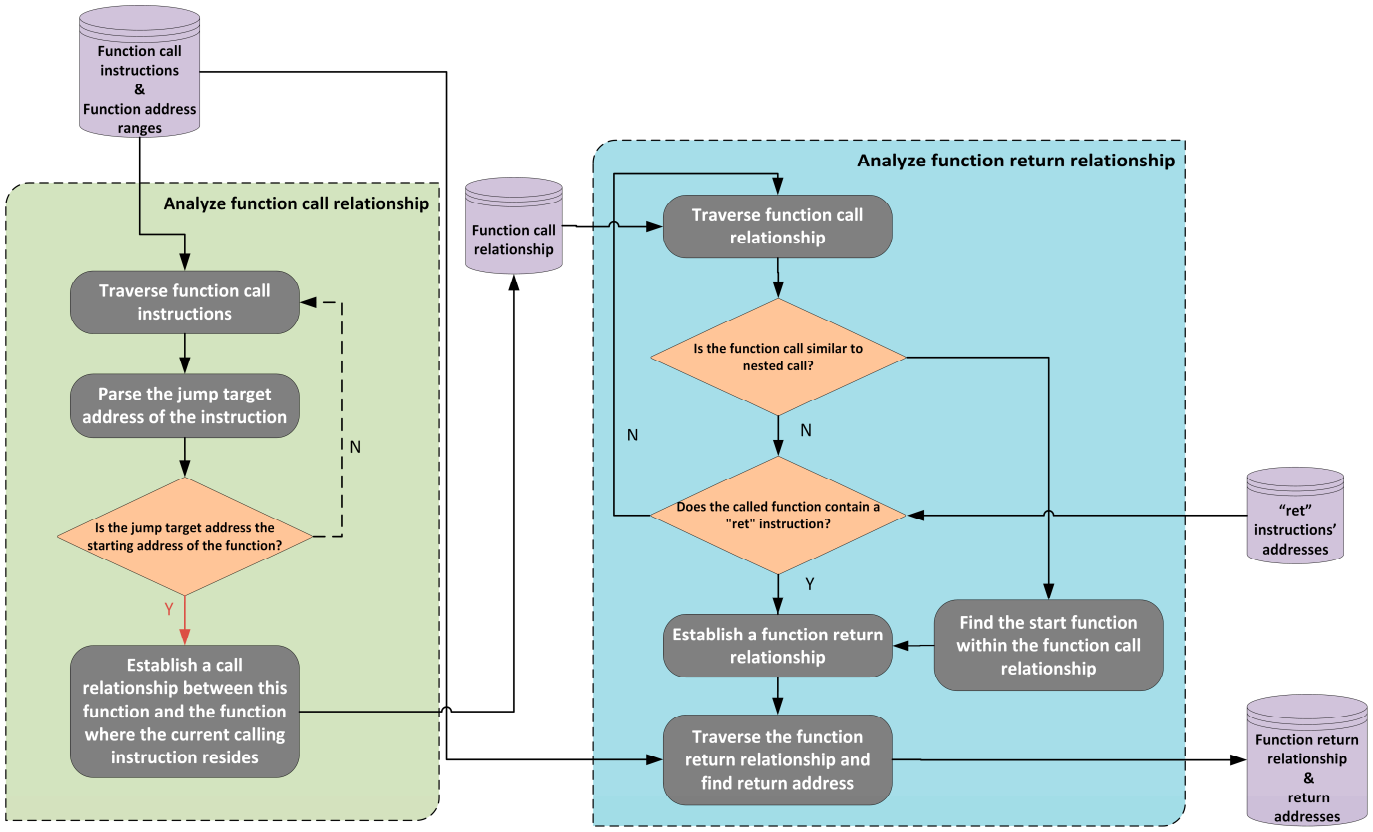


Figure 4. The analysis logic of CFIEE on function call and return.

It is worth noting that the analysis procedures in CFIEE are static, meaning that they do not account for dynamic changes or runtime behavior. This limitation results in CFIEE currently being unable to analyze the target address of indirect jumps, which may hinder its effectiveness in certain scenarios. CFIEE divides basic blocks based on control transfer relationships obtained from previous analyses and specific division rules specified in Table 4. During the division of basic blocks, we take into consideration the possibility of jump or branch target instructions within certain basic blocks. To address this, we have introduced two functions in the basic block division process: "create_basic_blocks_in_order()" and "create_basic_blocks_start_with_taken_target()". The first function strictly adheres to the basic block division rules, which is based on the disassembly file and the control transfer relationships derived from the previous analysis. It divides the basic blocks in accordance with the address order of the instructions. On the other hand, the second function, "create_basic_blocks_start_with_taken_target()", focuses specifically on creating a new basic block starting at an address where a jump or branch target instruction resides. This allows us to capture any potential changes in control flow caused by these instructions effectively. By executing these two functions, CFIEE is able to sort the basic blocks according to their starting addresses, ultimately providing accurate and comprehensive basic block information. The sorted basic block information, when combined with the subsequent generated CFG, enables researchers to effectively analyze the program's execution path and identify potential deadlock issues. Furthermore, through analysis of the program's loop structure, researchers can pinpoint loops that may cause performance bottlenecks and optimize them accordingly. Additionally, it helps to understand how different parts of the program interact.

Table 4. The division rules of basic blocks.

BB Edge	Rules
Begin edge	1. First instr. of any function 2. Target instruction of uncond. jumps or cond. branches 3. Instr. following any cond. branch or uncond. jump
End edge	1. Last instr. of any function 2. An uncond. jump instr. or a cond. branch instr. 3. The "ret" instruction

3.3.3. Data Curation and Output

The data sorting and output module of CFIEE gathers comprehensive information on basic blocks and computes their corresponding hash values. The calculation process accepts binary or hexadecimal instructions of the basic blocks as input, allowing users to select both the hash algorithm and the desired length of the resulting hash value. Currently, CFIEE offers four options for hash algorithms: MD5, SHA-1, SHA256, and SHA512. Users can select any of these algorithms based on their specific requirements. In addition to algorithm selection, CFIEE also allows the user to specify the length of the generated hash value. Available options include 8-bit, 16-bit, 32-bit, and custom length. This feature allows users to balance storage efficiency and accuracy according to their needs. Furthermore, we plan to enhance CFIEE by incorporating support for custom hash algorithms in future updates. In addition, CFIEE can effectively organize and output necessary control transfer instructions and functional information, providing researchers with comprehensive and accurate data information.

To simplify the process and enhance modularity, we encapsulate the main functions within the process into two different entities: "export_results()" and "generate_CFG()". Specifically, the "export_results()" function can systematically arrange data files and present them in a user-friendly text format. On the other hand, the "generate_CFG()" function plays a key role in building the control flow graph of a program, which provides researchers with a visual representation of the control flow in a RISC-V executable.

4. Application Scenarios of CFIEE

As a control flow static analysis tool, CFIEE can provide detailed and accurate data for the design and implementation of CFI mechanisms, especially CFG-based CFI. Researchers can develop suitable CFI mechanisms for RISC-V embedded devices through analysis results such as basic block information, control flow graphs, and the number of jump instructions within each function output by the tool. Below, we outline a straightforward method for utilization.

"xxx_control_transfer.bin" in Table 3 contains the forward jump instruction and the address information of the current instruction in binary form. Additionally, "xxx_bin_basic_block_info.txt" and "xxx_hex_basic_block_info.txt" contain binary and hexadecimal representations of basic block data alongside their respective hash values. Figure 5 shows the hardware circuit diagram of a basic CFI mechanism constructed using these data. In this mechanism, the hash values of basic blocks and the PCs corresponding to control transfer instructions are stored within designated registers. When the hash verification unit recognizes the last instruction of the basic block, it calculates the hash value of the current basic block and compares it with the pre-obtained hash value. If the results are the same, it proves that the instructions in the current basic block have not been tampered with. Simultaneously, the "Target Verification" unit in the CFI verification unit is responsible for comparing the PC of the control transfer instruction with its pre-analyzed target instruction. The CFI verification unit is equipped with registers for storing interrupt entry addresses and a shadow stack for validating function return addresses, ensuring the integrity of program interrupts and return addresses. Prior to entering the interrupt, the CFI verification unit examines whether the current interrupt entry address is stored in the register; if not, it is

considered an exception for interrupt entry address. During a function call, the program pushes the return address (RA) onto the main stack and updates the stack pointer (SP). Simultaneously, the CFI verification unit copies RA from the main stack to the shadow stack. Upon function return, before executing the return instruction, the program retrieves RA from the main stack and performs a return operation. However, prior to this execution of return instruction, the CFI verification unit validates RA against that on the shadow stack. If there is a match with RA on the shadow stack, it proceeds with normal return; otherwise, it identifies an abnormality in the return address. Any differences detected in the CFI verification unit imply potential tampering or alterations in the program's control flow.

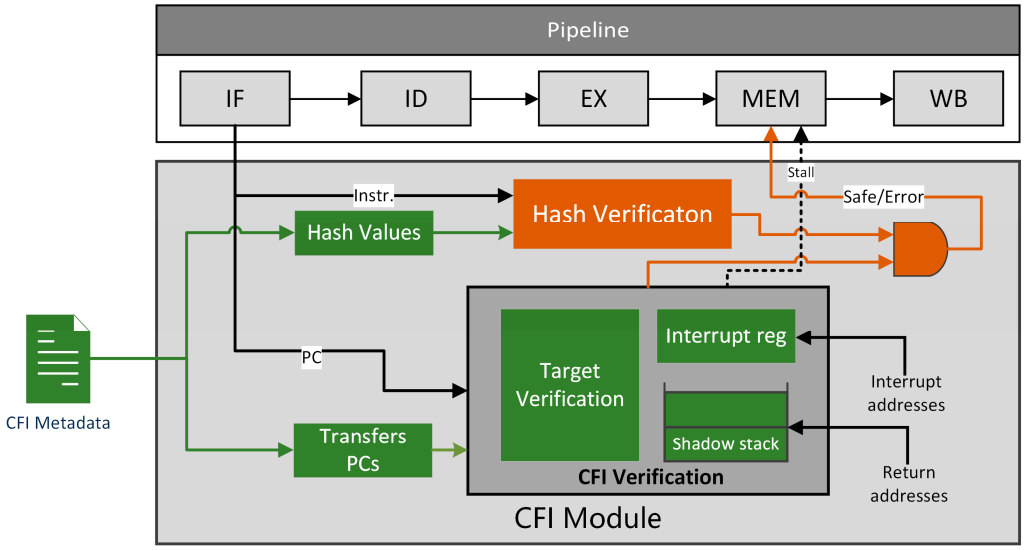


Figure 5. An example diagram of a CFI mechanism hardware circuit.

The verification of CFI mechanisms typically requires multiple cycles. To ensure a sufficient time margin for CFI verification, we have positioned the CFI mechanism between the IF and MEM phases in this work. The synchronization between the execution time of the processor, CFI verification module, user program, and other factors determines the alignment of CFI verification with program execution time. During the IF to MEM phase, it is possible for the EX and MEM phases to execute multiple cycles; in such cases, the running time of CFI verification may be shorter than that of the IF to MEM phase. If the verification process exceeds this pipeline section's running time, it necessitates a stall signal from the CFI verification unit to halt pipeline operation until completion of CFI verification. This is a trade-off between system performance and security

This article primarily offers a basic example of the Control Flow Integrity (CFI) mechanism without delving into specific CFI design intricacies. The main focus remains on providing an introductory illustration rather than exhaustive CFI design details. Future research will utilize the data files from CFIEE to craft CFI solutions suitable for RISC-V architecture. These efforts will delve deeper into CFI intricacies, aiming to create more specific and efficient CFI solutions tailored for the nuances of RISC-V architecture.

5. Evaluations

5.1. Comparison with Other Tools

Currently, there exist several control flow analysis tools available for the RISC-V architecture. For comparison purposes, our evaluation focuses on two specific tools: angr [9] and Turna [25]. Angr has garnered significant attention in the field of reverse engineering, and Turna's adoption of a hybrid approach enables it to generate a comprehensive Control Flow Graph (CFG). We have compiled a comparative analysis of their usability and capability to generate control flow information, as presented in Table 5.

Table 5. Comparison of CFIEE and other tools.

Features	CFIEE	angr [9]	Turna [25]
GUI	√	×	×
Without Extra Programming	√	×	√
Drawing CFG	√	√	√
Hash Calculation	√	√	×
Function Call Relationship	√	√	×

Among these three tools, CFIEE stands out as the only one with a GUI operation interface. As angr is a Python library, users need to write a Python program in order to invoke it for further analysis. Unlike angr, both CFIEE and Turna streamline user interaction by eliminating the necessity for users to write additional application programs. Regarding Control Flow Graph (CFG) output functionality, all three tools demonstrate the capability to generate outputs. Both CFIEE and angr can output the hash value of the program basic blocks and the calling relationship of the function. Notably, Turna, being primarily a CFG reconstruction tool, currently lacks these specific functionalities.

While angr and Turna were specifically chosen for comparison in this evaluation due to their usability and ability to generate control flow information, it is important to note that each tool has its own strengths and weaknesses depending on specific requirements or research objectives. We acknowledge the capabilities of angr in obtaining detailed program execution data through static analysis and simulation operations. We also appreciate Turna's idea of using a hybrid approach to rebuild CFG. However, the primary focus of CFIEE research remains centered on offering a straightforward and efficient approach to furnish precise and readily accessible metadata essential for the hardware-based Control Flow Integrity (CFI) mechanism in RISC-V embedded systems. CFIEE aims to provide crucial data, such as hash values of basic blocks, program control flow graphs, instruction jump relationships, and function calling connections. These data are easily and swiftly obtainable through the configuration of the Python environment and the RISC-V toolchain within CFIEE.

5.2. Functional Evaluation

For functional evaluation, we selected 15 programs from the Beebs benchmark [28]. In order to better test the functionality of CFIEE, we made some changes to the code of the test programs. We modified the initial "fputc" function to add the serial port output related function of the T-head Xuantie E906 processor. In terms of a tool chain, we used the Xuantie-900-gcc-elf-newlib-x86_64-V2.6.1 RISC-V tool chain. This tool chain retains the functions of the original RISC-V tool chain and adds optimization options for the T-Head processors. The test platform utilized was CentOS 7. As CFIEE is developed based on Python 3, our test environment employs Python version 3.11.0.

The quantitative test results of the selected programs are presented in Table 6. These results consist of two sets of data: the number of basic blocks and the number of forward transfer instructions. The count of basic blocks can partially reflect the program's complexity, while the count of forward transfer instructions can reflect the transfer frequency of the program control flow.

Table 6. Analysis results of selected programs in Beebs benchmark.

Program Name	Traversed Func.	Traversed Instr.	Basic Blocks	Edges	Forward Transfers
cover	40	5514	1632	2765	1287
crc	38	5587	1633	2768	1289
ctl_stack	19	477	108	166	66
dijkstra	40	5703	1647	2791	1298
duff	39	5567	1624	2749	1280
fir	14	267	49	70	30

Table 6. *Cont.*

Program Name	Traversed Func.	Traversed Instr.	Basic Blocks	Edges	Forward Transfers
insertsort	39	5561	1618	2744	1279
jfdcint	39	5750	1618	2745	1276
lcdnum	15	293	58	83	35
nettle_des	41	7283	1621	2743	1278
nettle_md5	40	6402	1636	2773	1292
qurt	41	5912	1725	2928	1363
rijndael	44	9725	1736	2931	1358
sglib_dllist	17	506	135	217	92
sglib_rbtree	19	614	155	244	113

Table 7 showcases one of the basic blocks present in the "basic_block.txt" file, which is the output of CFIEE. Each basic block's metadata include essential elements such as block number, label, entry address, length, all instructions, and two possible transfer targets. The block number serves as a unique identifier for each basic block within the program. The entry address indicates the starting point of the basic block within the program's memory space. Length refers to the size or extent of a particular basic block in terms of its instruction count. All instructions listed in each basic block's metadata provide a comprehensive overview of what operations are performed within that particular segment. In cases where the final instruction of a basic block is a conditional branch, there will be two transfer targets mentioned in its metadata. A conditional branch allows for decision making based on certain conditions being met or not met during program execution. The presence of two transfer targets signifies that control flow can diverge into two separate paths depending on whether those conditions are satisfied or not. On the contrary, when the final instruction in a basic block is an unconditional jump, it means that the control flow will directly transfer to another location without any condition being evaluated. In this scenario, the basic block associated with this jump will have only one target for transferring control. The absence of a second target implies that there is no alternative path or decision point to be considered after executing this particular instruction.

Table 7. Example of "basic_block.txt" file format.

One of the Basic Blocks in "basic_block.txt"
Basic_block Name: 48 **In Function:** <main> **Start address:** 2940 **End address:** 2940 **Start instruction:** 2940: fe941ae3 bne s0, s1, 2934 <main+0x110> **End instruction:** 2940: fe941ae3 bne s0, s1, 2934 <main+0x110> **Length:** 1 **Taken_Target address:** 2934 **Taken_Target instruction:** 2934: 00040513 mv a0, s0 **Not_Taken_Target address:** 2944 **Not_Taken_Target instruction:** 2944: 0000d2b7 lui t0,0xd **Instruction:** 2940: fe941ae3 bne s0, s1, 2934 <main+0x110>

The two figures in Figure 6 display the binary and hexadecimal representations of the basic block metadata. For both files, we consistently assigned the same data elements, including basic block numbers, binary or hexadecimal instructions and addresses, and hash values obtained from instructions and user settings. This standardization of data elements ensures uniformity and facilitates efficient analysis and comparison during the evaluation process.

```
Basic_block: 8
bin_basic_block_instructions:
        0000010111110010: 10100000101
        0000010111110100: 11100000101
        0000010111110110: 11111111111101110000011110010011
        0000010111111010: 11110110011001110110011
hash_value:
        11011100010011011001100110100010
```

(**a**)

```
Basic_block: 8
bin_basic_block_instructions:
        5f2: 0505
        5f4: 0705
        5f6: fff70793
        5fa: 00f66763
hash_value:
        dc4d99a2
```

(**b**)

Figure 6. (**a**) Block's metadata in binary form; (**b**) block's metadata in hexadecimal form.

Figure 7a presents an exemplar of forward control transfer instructions extracted by CFIEE. To streamline data analysis, we systematically categorize all forward control transfer instructions within the specified analysis range according to their corresponding functions, storing them in the data file generated by the tool. Pairing transfer instructions with their respective target instructions facilitates easier analysis and comparison. It is worth noting that when dealing with branch jump instructions, we specifically focus on storing only the target instruction when the branch is "taken". This approach helps us prioritize relevant information while avoiding unnecessary duplication or cluttering of data.

The binary metadata associated with the control transfers showcased in Figure 7a are provided in Figure 7b, which contains all addresses of forward transfers. Each line consists of a 32-bit binary number, where the initial 16 bits represent the binary address of the jump instruction, and the final 16 bits delineate the target address of the jump instruction. These binary data can be directly utilized by researchers in CFI solutions, such as being stored in the secure memory of hardware for utilization by hardware-based CFI mechanisms. In the current format, the hardware overhead caused by storing the data of this file into memory is

$$overhead(Bytes) = \frac{Forward_{transfers_{num}} * 32}{4} \quad (1)$$

The current binary file format is not specifically designed for a particular CFI mechanism, and researchers have the flexibility to modify its data format and volume according to their research requirements.

Furthermore, Figure 7c illustrates the count of transfer instructions per function across four selected programs. This visualization offers a comprehensive insight into the control flow behavior and distribution within the codebase, thereby enhancing researchers' understanding of the program's structural intricacies. By examining the number of transfer instructions per function, researchers can identify patterns and trends that reveal how information flows through different parts of the code.

Figure 8 illustrates an example of function call relationships generated by CFIEE. CFIEE analyzes function call relationships based on unconditional jump instructions within functions. In Figure 8, asterisk labels (*) are appended at the end of specific nodes, signifying functions reached through the 'j' instruction. This comprehensive representation aids in understanding the function call relationships and the flow of control within the codebase, incorporating both "jal" and "j" instructions to offer a more precise and detailed analysis.

The Control Flow Graph (CFG) serves as a crucial metadata for Control Flow Integrity (CFI), ensuring the output of a complete and accurate CFG was a primary objective during the development of CFIEE. In Figure 9, we present a portion of the control flow graph obtained for the "lcdnum" program. Some basic blocks are labeled with "start with taken target" at the end of their names, indicating that the start address of the basic block serves as the target address of a control transfer instruction. The solid black arrows in Figure 9 represent unconditional jumps and "taken" branches resulting from branch jumps, while the red dotted arrows indicate branches of branch jumps that are not taken.

Additionally, in certain basic blocks, a combination of function name and address may appear in the "Taken target" column. This labeling signifies the target address specifically

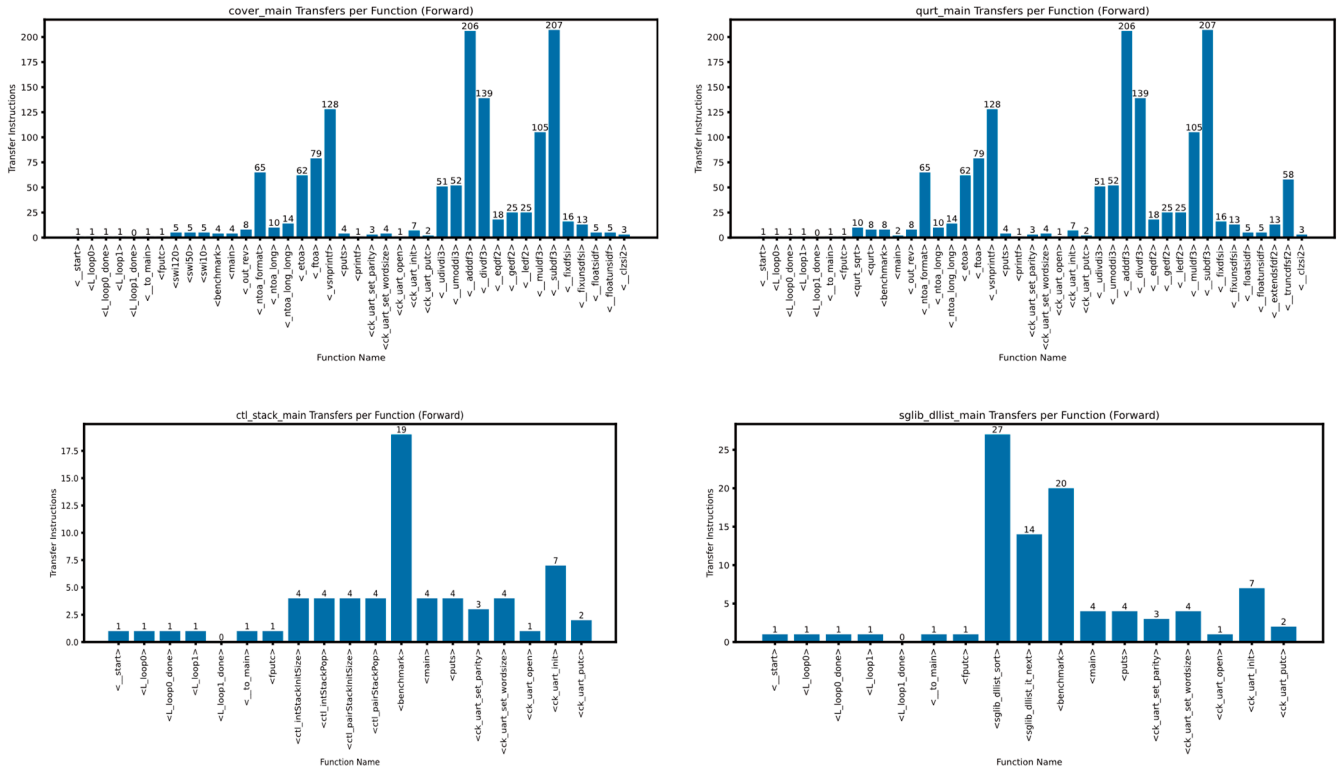
designated for the ret instruction. Since a function may be called by different functions at various times, the ret instruction within a function may have multiple return target addresses. To facilitate researchers in analyzing the ret instruction, we include all target addresses and corresponding functions in the "Taken target" line. This comprehensive representation of the CFG through CFIEE enhances the analysis of control flow integrity and provides valuable support for researchers in understanding the intricacies of the codebase.

```
<benchmark>:

j/b_instr: 5c8: 00e6fa63                bgeu    a3,a4,5dc <benchmark+0x28>
t_instr:   5dc: 0585                    addi    a1,a1,1

j/b_instr: 5d8: fee6eae3                bltu    a3,a4,5cc <benchmark+0x18>
t_instr:   5cc: c398                    sw      a4,0(a5)

j/b_instr: 5e0: fea590e3                bne     a1,a0,5c0 <benchmark+0xc>
t_instr:   5c0: 4214                    lw      a3,0(a2)
```

(**a**)

```
00000101110010000000010111011100
00000101110110000000010111001100
00000101111000000000010111000000
```

(**b**)



(**c**)

Figure 7. (**a**) An example of metadata for control transfer instructions; (**b**) binary metadata associated with control transfer instructions in (**a**); (**c**) the number of transfer instructions per function in four selected programs.

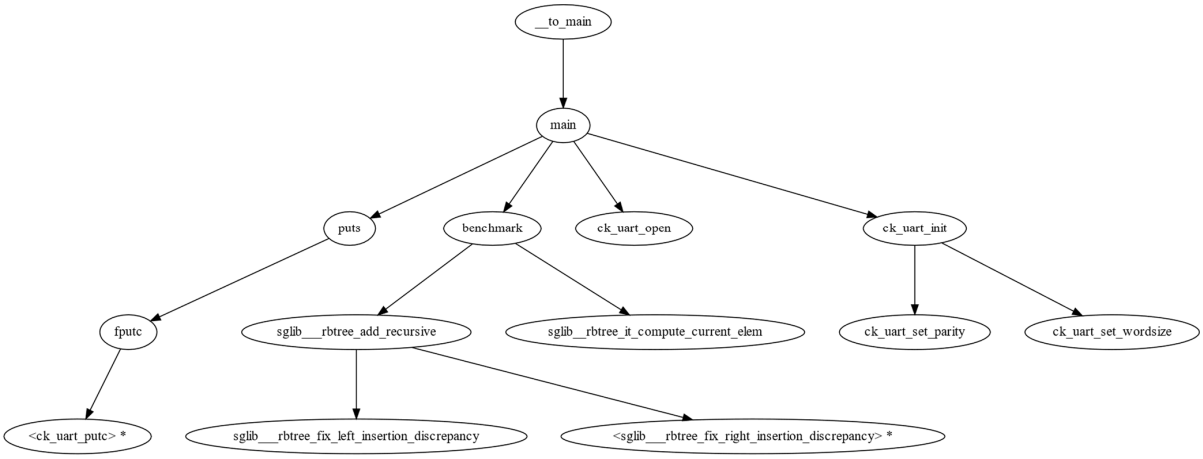


Figure 8. An example of the function call relationship output by CFIEE.

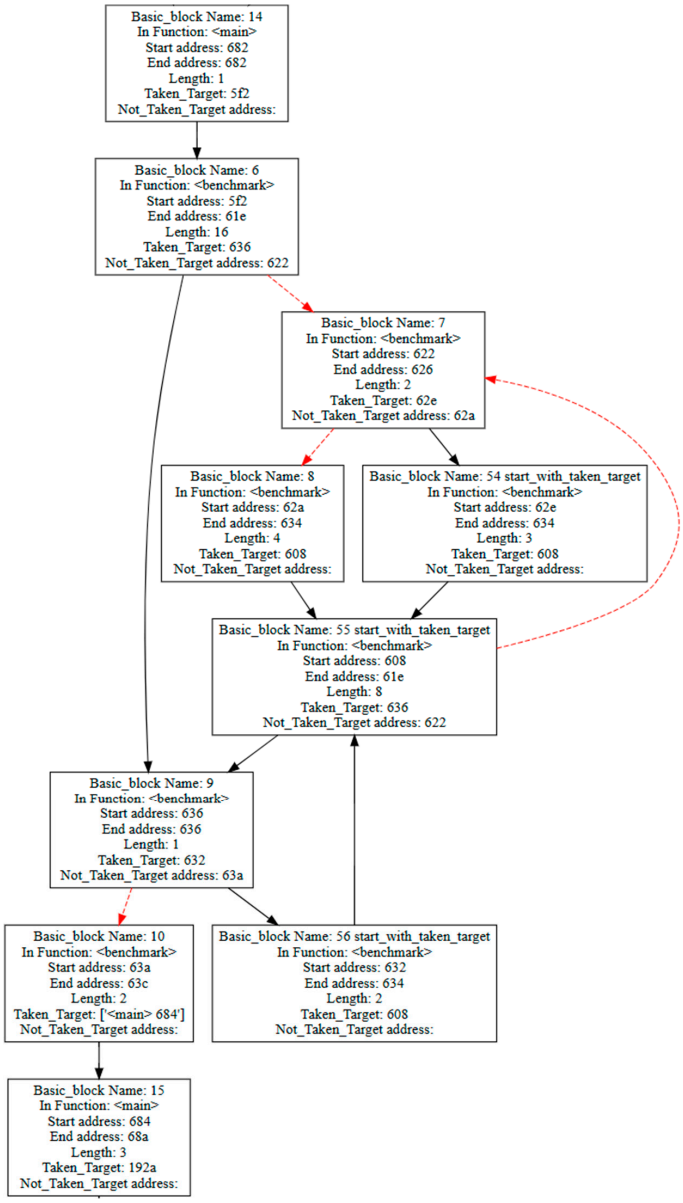


Figure 9. Part of "lcdnum" CFG.

6. Conclusions

In this paper, we present CFIEE, an open-source critical metadata extraction tool designed for enhancing hardware-based CFI research in the RISC-V architecture. CFIEE implements automatic static analysis of the control flow of RISC-V executable files, significantly lowering the usage threshold with its graphical interface operation. Researchers can utilize the program control flow graph, program basic block information, and other data output by CFIEE to analyze potential deadlocks, loop exceptions, and other issues within a given program. Furthermore, CFIEE offers valuable metadata for research on hardware-based CFI mechanisms that can aid in the development of secure and effective RISC-V control flow protection mechanisms.

This software simplifies the extraction of critical metadata and automates control flow analysis, reducing the burden of manual data extraction tasks. This increase in efficiency allows researchers to focus more on in-depth analysis and experimentation, ultimately designing more efficient CFI mechanisms that better secure RISC-V devices. The visualization of control flow metadata by CFIEE provides researchers with an accurate depiction of complex control flow relationships, facilitating quicker comprehension and validation of research findings.

While CFIEE currently offers a relatively comprehensive set of functions, there are still opportunities for improvement in terms of operational performance and scope of application. CFIEE currently lacks the capability to handle forward register-related jumps due to its static analysis nature [29]. However, it does possess corresponding analysis logic for indirect jumps of the "ret" type.

Since the initial presentation of this work [30], we aim to delve into indirect control flow analysis. On the software front, our plan involves integrating CFIEE with RISC-V compatible simulators to utilize simulation execution data for enhancing static analysis. Additionally, we intend to embark on a combined static–dynamic analysis approach. Regarding research into mechanisms, we will leverage existing lightweight hardware protection mechanisms [31] and integrate CFIEE's data support to investigate a more secure and efficient hardware-based RISC-V CFI mechanism. Furthermore, the metadata utilized in the hardware CFI mechanism has the potential for additional compression [32].

CFIEE is an open-source tool released under an open license, and we encourage users to extend and enhance its capabilities.

Author Contributions: W.L. and W.W. wrote this paper; W.W. conceived the proposed scheme and reviewed the manuscript; W.L. designed the proposed software and the experiments; S.L. joined in the data analysis and discussion phases. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Natural Science Foundation of China (No. 62201325), the Science and Technology Support Plan for Youth Innovation of Colleges and Universities in Shandong Province of China (No. 2023KJ096), and the Shandong Provincial Natural Science Foundation (No. ZR2020QF027).

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Lhee, K.S.; Chapin, S.J. Buffer overflow and format string overflow vulnerabilities. *Softw. Pract. Exper.* **2003**, *33*, 423–460. [CrossRef]
2. Roemer, R.; Buchanan, E.; Shacham, H.; Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur. (TISSEC)* **2012**, *15*, 1–34. [CrossRef]
3. Abadi, M.; Budiu, M.; Erlingsson, U.; Ligatti, J. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur. (TISSEC)* **2009**, *13*, 1–40. [CrossRef]

4. Mishra, T.; Chantem, T.; Gerdes, R. Survey of Control-flow Integrity Techniques for Real-time Embedded Systems. *ACM Trans. Inf. Syst. Secur. (TISSEC)* **2022**, *21*, 1–32. [CrossRef]
5. Dariz, L.; Ruggeri, M.; Selvatici, M. A static microcode analysis tool for programmable load drivers. In Proceedings of the 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), Bremen, Germany, 27–28 September 2015; pp. 265–270. [CrossRef]
6. Almossawi, A.; Lim, K.; Sinha, T. *Analysis Tool Evaluation: Coverity Prevent*; Carnegie Mellon University: Pittsburgh, PA, USA, 2006; pp. 7–11.
7. Nethercote, N.; Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Not.* **2007**, *42*, 89–100. [CrossRef]
8. Luk, C.-K.; Cohn, R.; Muth, R.; Patil, H.; Klauser, A.; Lowney, G.; Wallace, S.; Reddi, V.J.; Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Not.* **2005**, *40*, 190–200. [CrossRef]
9. Wang, F.; Shoshitaishvili, Y. Angr—The Next Generation of Binary Analysis. In Proceedings of the 2017 IEEE Cybersecurity Development (SecDev), Boston, MA, USA, 24–26 September 2017; pp. 8–9.
10. Cadar, C.; Dunbar, D.; Engler, D.R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the OSDI, San Diego, CA, USA, 8–10 December 2008; pp. 209–224.
11. T-Head Xuantie E906 Datasheet. Available online: https://www.xrvm.cn/product/xuantie/E906 (accessed on 25 March 2024).
12. Waterman, A.; Asanovi, K. (Eds.) *The RISC-V Instruction Set Manual*; Volume I: Unprivileged ISA, Document Version 20190608-Base-Ratified; RISC-V International: Zurich, Switzerland, 2019.
13. Waterman, A.; Asanovi, K. (Eds.) *The RISC-V Instruction Set Manual*; Volume II: Privileged Architecture Document Version 20190608-Priv-MSU-Ratified; RISC-V International: Zurich, Switzerland, 2019.
14. CFIEE: A Critical Metadata Extraction Engine for RISC-V Hardware CFI Scheme. Available online: https://github.com/Taurus052/CFIEE (accessed on 25 March 2024).
15. Kanter, D. RISC-V offers simple, modular ISA. *Microprocess. Rep.* **2016**, *1*, 1–5.
16. Waterman, A.; Lee, Y.; Patterson, D.; Asanovic, K. *The RISC-V Instruction Set Manual*; Volume I: User-Level ISA', Version 2.0; RISC-V International: Zurich, Switzerland, 2014.
17. Allen, F.E. Control flow analysis. *ACM Sigplan Not.* **1970**, *5*, 1–19. [CrossRef]
18. Jing, J.; Jiang, L.-H.; Liu, T.-M.; Wang, Z.-Y.; Wang, R.-M. A precision-tunable CFG reconstruction algorithm. In Proceedings of the 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC), Shenyang, China, 20–22 December 2013; pp. 2095–2099. [CrossRef]
19. Jang, H.; Park, M.C.; Lee, D.H. IBV-CFI: Efficient fine-grained control-flow integrity preserving CFG precision. *Comput. Secur.* **2020**, *94*, 101828. [CrossRef]
20. Park, M.C.; Lee, D.H. BGCFI: Efficient Verification in Fine-Grained Control-Flow Integrity Based on Bipartite Graph. *IEEE Access* **2023**, *11*, 4291–4305. [CrossRef]
21. Niu, B.; Tan, G. Per-Input Control-Flow Integrity. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 914–926. [CrossRef]
22. Yin, W.; Jiang, L.; Yin, Q.; Zhou, L.; Li, J. A control flow graph reconstruction method from binaries based on XML. In Proceedings of the 2009 International Forum on Computer Science-Technology and Applications, Chongqing, China, 25–27 December 2009; pp. 226–229. [CrossRef]
23. Yount, C.; Patil, H.; Islam, M.S.; Srikanth, A. Graph-matching-based simulation-region selection for multiple binaries. In Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, USA, 29–31 March 2015; pp. 52–61. [CrossRef]
24. Barbar, M.; Sui, Y.; Zhang, H.; Chen, S.; Xue, J. Live path control flow integrity. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, Gothenburg, Sweden, 30 May–1 June 2018; pp. 195–196. [CrossRef]
25. Sahin, V.H. Turna: A control flow graph reconstruction tool for RISC-V architecture. *Computing* **2023**, *105*, 1821–1845. [CrossRef]
26. Li, S.; Wang, W.; Li, W.; Zhang, D. Hardware-Based Software Control Flow Integrity: Review on the State-of-the-Art Implementation Technology. *IEEE Access* **2023**, *11*, 133255–133280. [CrossRef]
27. Tauner, S.; Telesklav, M. Comparative analysis and enhancement of CFG-Based hardware-assisted cfi schemes. *ACM Trans. Embed. Comput. Syst. (TECS)* **2021**, *20*, 1–25. [CrossRef]
28. Pallister, J.; Hollis, S.; Bennett, J. BEEBS: Open benchmarks for energy measurements on embedded platforms. *arXiv* **2013**, arXiv:1308.5174. [CrossRef]
29. Burow, N.; Carr, S.A.; Nash, J.; Larsen, P.; Franz, M.; Brunthaler, S.; Payer, M. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* **2017**, *50*, 1–33. [CrossRef]
30. Li, W.; Wang, W.; Li, S.; An, Z. A Static CFG Extraction Scheme for RISC-V Runtime CFI. In Proceedings of the 9th International Symposium on System Security, Safety, and Reliability (ISSSR 2023), Hangzhou, China, 10–11 June 2023; pp. 444–445. [CrossRef]

31. An, Z.; Wang, W.; Li, W.; Li, S.; Zhang, D. Securing Embedded System from Code Reuse Attacks: A Lightweight Scheme with Hardware Assistance. *Micromachines* **2023**, *14*, 1525. [CrossRef] [PubMed]
32. Kanuparthi, A.; Rajendran, J.; Karri, R. Controlling your control flow graph. In Proceedings of the 2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, VA, USA, 3–5 May 2016; pp. 43–48. [CrossRef]