Hetrify: Efficient Verification of Heterogeneous Programs on RISC-V

Yiwei Li, Liangze Yin*, Wei Dong*, Jiaxin Liu, Yanfeng Hu, Shanshan Li
National University of Defense Technology, China
hn.cs.lyw@163.com, (yinliangze, wdong, liujiaxin18, huyanfeng22, shanshanli)@nudt.edu.cn

***Abstract*—The heterogeneous nature of contemporary software, comprising components like closed-source libraries, embedded assembly snippets, and modules written in multiple programming languages, leads to significant verification challenges. Currently, there are no mature and available methods to effectively address such problems. To bridge this gap, we propose a verification approach capable of effectively verifying heterogeneous programs. This approach is universally applicable. It theoretically supports the verification of any heterogeneous program that can be compiled into binary code, without being constrained by any specific programming language. The approach begins by compiling the entire program or its unverifiable segments into binary format. Under guarantees of semantic equivalence, these binaries are converted into verifiable C code, which can then be verified using existing C verification tools. Based on the RISC-V architecture, we developed the Hetrify tool to implement this verification approach. The tool is supported by rigorous mathematical proofs to ensure operational semantic equivalence between the converted C programs and their original counterparts. To validate our approach, we conducted verification experiments on 130 programs, including 100 assembly programs and 30 large heterogeneous programs with missing critical function source code, demonstrating the effectiveness of our approach.**

***Index Terms*—RISC-V, Heterogeneous program, Program verification**

I. Introduction

In contemporary software development, programs are often collaboratively developed by multiple teams, with different modules potentially written in various programming languages [1], [2]. This heterogeneity is common in complex systems, where each module is designed to leverage the strengths of a particular language [3], [4]. However, the source code for essential third-party library functions, which might be written in different languages, is often inaccessible, posing significant challenges for program verification [5]. When the source code is composed of multiple programming languages, critical functions are unavailable, or there are no mature automated verification tools, verification processes can become inaccurate [6]. This inadequacy compromises the reliability and security of the software.

Previous research has explored various strategies for program verification, focusing on abstracting the behavior of programs where source code cannot be directly verified. Tools like SLAM [7] and CIL [8] abstract program behaviors to perform detailed source code checks. Some methods decompile binary code of unverifiable segments into verifiable code forms. However, tools like BinDiff [9] and IDA Pro [10] often fail in conversion, and even when successful, produce pseudocode that cannot be directly verified. Recently, AI techniques using neural machine translation models [11] and large language models [12] have been applied to decompile binary code to high-level code. While promising, these approaches focus on code similarity rather than semantic equivalence [13].

Other studies have focused on verifying mixed-language programs. Schmaltz and Shadrin proposed a method to verify mixed low-level language programs by ensuring compatibility with MASM and the Windows ABI. Still, this approach is limited by its dependence on specific ABI standards, reducing flexibility [14]. Recoules et al. introduced TInA, which converts inline assembly into verifiable C code for x86 and ARM architectures. While effective, TInA is limited to small-scale code and struggles to maintain full semantic equivalence after conversion [15].

Inspired by [15]–[17], this paper proposes a verification approach that converts unverifiable segments of heterogeneous programs into verifiable C code while ensuring semantic equivalence. These segments may include low-level language code, closed-source library functions, and code written in other programming languages. The converted programs are then integrated with the original program for comprehensive verification. To implement this approach, we chose the RISC-V architecture due to its widespread adoption in academia and industry, the potential for future development, and ability to meet diverse application needs through its base instruction set and optional extensions [18], [19]. Based on this architecture, we developed Hetrify, a tool designed to verify heterogeneous programs. Hetrify theoretically allows for the automated verification of any compiled language program, as long as it can be compiled into RISC-V binary files, which Hetrify can then convert into C code for automated verification.

To ensure the converted C programs are semantically equivalent to the original code, we mathematically proved that the operational semantics of the original code's compiled binary are equivalent to those of the converted C code. We validated our method with experiments involving 100 assembly programs and large heterogeneous programs with missing critical function source code. The results showed that all 130 programs were accurately verified, with incorrect programs

*Corresponding authors: Liangze Yin, Wei Dong

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00081

TABLE I: RISC-V Instruction Categories and Corresponding Instructions

Category	Instructions
Load & Store	lb, lh, lw, ld, lbu, lhu, lwu, sb, sh, sw, sd
Jump	jal, jalr
Conditional Branch	beq, bne, blt, bge, bltu, bgeu
Immediate	auipc, lui
Immediate-Based Computation	addi, addiw, andi, ori, xori, slli, slliw, slti, sltiu, srai, sraiw, srli, srliw
Register-Based Computation	add, addw, sub, subw, rem, remu, remw, remuw, mul, mulw, div, divu, divw, divuw, and, or, xor, sll, sllw, slt, sltu, sra, sraw, srl, srlw

identified and correct ones verified, except for a few timeouts in the assembly programs. This indicates that, given sufficient verification time and computational resources, our method can achieve 100% verification accuracy, demonstrating both its effectiveness and the correctness of our theoretical analysis.

To summarize, the key contributions of this paper are outlined as follows:

- We propose a novel verification approach for heterogeneous programs that enables the conversion of entire programs or their unverifiable segments into verifiable C code, ensuring operational semantic equivalence. This approach theoretically supports comprehensive verification of any compiled or low-level language program, even without source code access.
- To ensure the correctness of the verification approach, we mathematically demonstrated that the converted C programs maintain operational semantic equivalence with the original code.
- We developed Hetrify, a verification tool based on the RISC-V architecture and our proposed approach. Evaluations of 100 assembly programs and 30 large C programs with missing critical functions confirmed the effectiveness and theoretical correctness of our approach.

The rest of the paper is organized as follows. Section II presents the preliminary knowledge. Section III outlines the research motivation with an example program, illustrating the challenges for program verification. Section IV introduces the verification methodology and its workflow. Section V details the methods used to convert heterogeneous programs into verifiable C code. Section VI describes our experimental evaluation. Section VII discusses related work in the field. Section VIII discusses the limitations of this work and outlines future research directions. Finally, Section IX summarizes our findings and conclusions.

II. Preliminaries

A. RISC-V Instruction Set

RISC-V is an open-source instruction set architecture known for its simplicity and modularity [18]. It is widely used in applications ranging from small embedded systems to large data centers, gaining significant attention in academia and industry due to its flexibility [20]. RISC-V includes 4-byte base instructions and 2-byte compressed instructions to reduce code size and improve execution efficiency [18].

This study converts 59 RISC-V instructions listed in Table I into C code, categorizing them into five major groups. This covers most of the RISC-V base instruction set (RV64I), includes some multiply or divide extension instructions, and supports compressed instructions, meeting most program verification needs. Although similar methods apply to floating-point extension instructions, this study omits them for now. Due to limited support for system calls in current C verification tools, system call-related and privileged instructions are also excluded.

B. CBMC: C Bounded Model Checker

CBMC is an automated verification tool for static analysis of C codebases [21], [22]. It checks for array bounds, pointer safety, exceptions, and user-specified assertions. Using SAT or SMT solvers, CBMC verifies whether code assertions can be violated or if certain properties hold under specified conditions [23]. Notably, CBMC has excelled in the annual SV-COMP (Software Verification Competition), consistently performing well across various categories [24], [25]. In this study, CBMC was selected as the verification tool for experimental evaluation.

C. Program State Representation

Program execution can be represented as transitions between program states [26]. To accurately describe the execution of assembly programs and their C translations, we define the program state as follows.

Definition 1. The program state is represented as a quadruple:

$$\sigma = (Reg, Mem, FCS, pc),$$

where $Reg = \{x_0, x_1, \ldots, x_{31}\}$ defines the state of 32 general-purpose registers, each storing a specific 64-bit value. Each register x_i is represented as a pair (i, v_i), where i is the register identifier and v_i is the value stored in the register. For example, $x_1 = (1, \texttt{0x1})$ means that register x_1 holds the 64-bit value `0x1`.

The memory state Mem is a mapping of addresses to byte values, represented as pairs $(Addr, v)$, where $Addr$ is a memory address and v is the 8-bit value stored at that address. For instance, $Mem = \{(\texttt{0x1000, 0xFF}), (\texttt{0x1001, 0x1})\}$.

The function call stack state FCS tracks return addresses from function calls. The sets $\mathcal{P}(Reg)$, $\mathcal{P}(Mem)$, and

$\mathcal{P}(FCS)$ represent all possible states of the registers, memory, and function call stack, respectively.

Definition 2. The following functions are defined to operate on the program state.

- The functions $\mathcal{R}_r$ and $\mathcal{W}_r$ are defined to read from and write to registers, respectively:

$$\mathcal{R}_r : Reg \to \mathbb{N}, \quad \mathcal{W}_r : Reg \times \mathbb{N} \to \mathcal{P}(Reg),$$

  where $\mathbb{N}$ denotes the set of all natural numbers. Notably, $\mathcal{R}_r(x_0) = 0$ and $\mathcal{W}_r(x_0, n) = Reg$, meaning that reads from x_0 always return 0, and writes to x_0 do not change the register state.
- The functions $\mathcal{R}_m$ and $\mathcal{W}_m$ are defined to read from and write to memory, respectively:

$$\mathcal{R}_m : Mem \times Size \to \mathbb{N},$$
$$\mathcal{W}_m : Mem \times \mathbb{N} \times Size \to \mathcal{P}(Mem),$$

  where $Size = \{1, 2, 4, 8\}$ represents the number of bytes to be read from or written to memory by an instruction.
- The function $\mathcal{T}$ maps memory addresses to tags, representing the global variable or function associated with each address:

$$\mathcal{T} : Mem_t \to Tag,$$

  where $Tag = Tag_v \cup Tag_f$ denotes the names for global variables Tag_v and functions Tag_f, and $Mem_t \subseteq Mem$ includes the starting addresses of these variables and functions. $\mathcal{T}$ is a bijection, meaning each memory address is uniquely associated with a tag, and vice versa.
- The functions $push$, pop, and top operate on the FCS:

$$push : Addr \to \mathcal{P}(FCS),$$
$$pop : \mathcal{P}(FCS) \to \mathcal{P}(FCS),$$
$$top : \mathcal{P}(FCS) \to Addr,$$

  where the $push$ function adds the memory address corresponding to a function onto the stack, the pop function removes the function address from the top of the stack, and the top function returns the function address at the top of the stack.
- The semantic function $\mathcal{S}[\![\cdot]\!]$:

$$\mathcal{S}[\![\cdot]\!] : \text{Act} \to (\Sigma \hookrightarrow \Sigma),$$

  where Σ represents the set of all possible program states, Act represents the union of the set of assembly instructions and the set of C instructions.

D. Assembly Conversion Language

To present the conversion methodology and prove the equivalence between assembly programs and their converted C counterparts, this subsection defines a subset of the C language called the Assembly Conversion Language (ACL). Definitions 3 and 4 provide the syntax and operational semantics of ACL, respectively, considering only the case where each assembly instruction is 4 bytes.

Definition 3. The syntax of commands C and inner commands IC in the ACL is defined using the following BNF notation:

$$C ::= \texttt{L\_}pc : \{IC\}$$
$$IC ::= x_r := e; \mid *(type*)(e_1) := e_2; \mid \text{skip}; \mid IC_1; IC_2$$
$$\mid \text{if } (e)\{IC\} \mid \text{goto L\_}pc; \mid f(); \mid return;$$

Operators op and expressions e and variable type $type$ are given by

$$op ::= \mid == \mid != \mid < \mid >= \mid \ \% \ \mid \ * \ \mid \ + \ \mid \ - \ \mid \ /$$
$$e ::= x_r \mid x_v \mid \&x_v \mid n \mid e \ op \ e \mid !(e) \mid (type)e \mid *(type*)e$$
$$type ::= \text{char} \mid \text{unsigned char} \mid \text{short} \mid \text{unsigned short}$$
$$\mid \text{int} \mid \text{unsigned int} \mid \text{long} \mid \text{unsigned long}$$

The syntax of $n \in \mathbb{N}$ denotes integer numbers, $f()$ represents a function call with $f \in Tag_f$, $\texttt{L\_}pc$ represents a label in C program, where pc denotes the address of the current assembly instruction. The variables x include register variables x_r (simulating registers) with $r \in \{0, 1, 2, \ldots, 31\}$ and $x_v \in Tag_v$ representing global variables. The address of a global variable is given by $\&x_v = \mathcal{T}^{-1}(x_v)$.

Definition 4. The operational semantics of commands C and inner commands IC in the assembly conversion language are defined as follows. The transition relation $\mathcal{S}[\![C]\!]$ specifies how a command C changes the state $\sigma = (Reg, Mem, FCS, pc)$.

$$\mathcal{S}[\![\texttt{L\_}pc : \{IC\}]\!](\sigma) = \mathcal{S}[\![IC]\!](Reg, Mem, FCS, next(pc))$$
$$\mathcal{S}[\![x_r := e;]\!](\sigma) = (\mathcal{W}_r(x_r, e), Mem, FCS, pc)$$
$$\mathcal{S}[\![\texttt{skip};]\!](\sigma) = \sigma$$
$$\mathcal{S}[\![IC_1; IC_2]\!](\sigma) = \mathcal{S}[\![IC_2]\!](\mathcal{S}[\![IC_1]\!](\sigma))$$
$$\mathcal{S}[\![\text{if } (e)\{IC\}]\!](\sigma) = \begin{cases} \mathcal{S}[\![IC]\!](\sigma) & \text{if } e \neq 0 \\ \sigma & \text{otherwise} \end{cases}$$
$$\mathcal{S}[\![\texttt{goto L\_}\hat{pc};]\!](\sigma) = (Reg, Mem, FCS, \hat{pc}))$$
$$\mathcal{S}[\![f();]\!](\sigma) = (Reg, Mem, push(pc + 4), \mathcal{T}^{-1}(f))$$
$$\mathcal{S}[\![\texttt{return};]\!](\sigma) = (Reg, Mem, pop(FCS), top(FCS))$$
$$\mathcal{S}[\![*(type*)(e_1) := e_2;]\!](\sigma) = (Reg, W, FCS, pc)$$

where $W = \mathcal{W}_m\,(e_1, e_2, \text{sizeof}(type))$, with sizeof calculating the size of the specified variable type. $\hat{pc}$ represents the possible values of pc. Moreover, $next(pc)$ adjusts the pc after command execution. If the command includes a `goto`, function call, or `return` statement, $next(pc)$ is set by the target address specified in the inner command. Otherwise, $next(pc)$ increments by 4, setting it to $pc + 4$.

III. Motivation

In real-world software verification, the complexity increases significantly when source code is composed in a complex manner, such as when the source code for crucial functions is inaccessible, or when the program is written in multiple languages. Acknowledging these challenges, this section outlines the study's core objectives and approach. It also illustrates the research motivation using an example program, presents strategies for addressing verification challenges.

```
1 #include <assert.h>
2 extern int __VERIFIER_nondet_int();
3 extern void insertion_sort(int*, unsigned int);
4 int main() {
5     unsigned int SIZE = 10;
6     int v[10];
7     for (int j = 0; j < SIZE; j++) {
8         v[j] = __VERIFIER_nondet_int();
9     }
10    insertion_sort(v, SIZE);
11    for (int k = 1; k < SIZE; k++) {
12        assert(v[k - 1] <= v[k]);
13    }
14    return 0;
15 }
```

Fig. 1: Motivational example

To effectively illustrate the motivation for this study, we use a small example program to elucidate the study's core objectives and approach. The program in Fig. 1, adapted from the `insertion_sort-1.c` verification program used in SV-COMP 2024 [27], aims to verify the correctness of the `assert` statement on line 12. The function `__VERIFIER_nondet_int` returns a random signed integer. In this program, the insertion sort algorithm is verified, indicating that the correctness of the `assert` statement heavily depends on the `insertion_sort(v, SIZE);` call on line 10. However, the source code for the `insertion_sort` function is not provided or is written in another language, reflecting a common scenario in real-world software development where developers frequently rely on library functions available as static or dynamic libraries without accessible or consistent source code.

It is particularly challenging to verify heterogeneous programs that include function calls without accessible source code or contain segments written in other languages. Typically, current verification tools treat such functions as uninterpreted functions, leading to inaccurate verification results. For example, in Fig. 1, the `insertion_sort` function would be treated as an uninterpreted function, which misrepresents its behavior and leads to incorrect verification of the `assert` statement on line 12.

To verify these heterogeneous programs, we propose a verification approach that converts their source code into verifiable C code, which is then verified using existing mature C verification tools. The main challenge is to ensure that this conversion does not alter the programs' operational semantics, maintaining the original behavior for accurate and reliable verification.

IV. Method Overview

In this section, we present a verification approach for heterogeneous programs by converting them into verifiable C code. This approach involves three main steps: source code compiling, ELF file analysis, and binary code conversion, as shown in Fig. 2. To clearly describe the workflow of this verification approach, we use the insertion sort algorithm example from Fig. 1. Although the example provided is specific to C, the verification approach is theoretically applicable to any compiled language, so please disregard any language-specific limitations.

A. Source Code Compiling

Considering the absence of dedicated verification tools for some high-level languages, we need to adopt a method that compiles the entire program into a binary file and then converts it into a verifiable C program to automate the verification process. This source code compiling phase involves two main steps: preprocessing the source code and compiling the preprocessed code into binary object files.

For high-level language programs, verification properties (specifications) are typically embedded as assertions. However, directly compiling these assertions into binary code can generate excessive irrelevant code, complicating verification. Therefore, during preprocessing, the main goal is to perform semantically equivalent conversions on these specifications. This involves converting assert statements to a form that simplifies verification and removing unnecessary functions like `__VERIFIER_nondet_int` to avoid extra code generation during compilation. For example, in Fig. 1, the assertion `assert(v[k - 1] <= v[k])` on line 12 is transformed into `if(!(v[k - 1] <= v[k])) {assert_ret1 = 0;}`, allowing verification by simply checking if the global variable `assert_ret1` equals 1.

After preprocessing, the code is compiled into object files using the appropriate compiler for the program's language. A linker and a linker script are then used to link the object files into an ELF file. The linker script ensures the object files are loaded to the correct target addresses, even when the source code to be verified does not contain a main function. At this stage, the first phase has been successfully completed, resulting in the generation of an ELF file through preprocessing, compilation, and linking.

B. ELF File Analysis

This phase involves two steps. First, the ELF file is analyzed to extract and record all global variables and function entry addresses. Second, the program is loaded into the simulator's memory, and the initial values of all global variables are recorded from the simulator's memory. Using this information, the initial framework of the verifiable C program is constructed, including global variable declarations and initializations, function declarations, and related structures.

As shown in Fig. 3, after the ELF file analysis phase, a verifiable C code framework is obtained, as illustrated in Fig. 4. This framework defines essential data structures for program execution. For instance, lines 4 to 7 define the 32 general-purpose registers and the program counter of the RISC-V processor. Line 8 defines the global variables, with initial values, types, and names derived from the ELF file. Lines 16, 20, and 21 define the stack space used during program execution, with `STACK_SIZE` calculated as the sum of the stack sizes used by each function. The framework also includes existing function declarations, as shown in lines 9 to 10. To

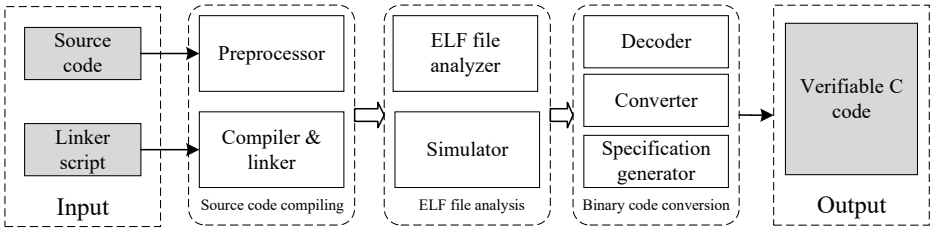

Fig. 2: Overview of verification approach

```c
int assert_ret1=1;
extern void insertion_sort(int*, unsigned int);
int main() {
    unsigned int SIZE = 10;
    int v[10];
    for (int j = 0; j < SIZE; j++) {
        v[j];
    }
    insertion_sort(v, SIZE);
    for (int k = 1; k < SIZE; k++) {
        if(!(v[k - 1] <= v[k])) {assert_ret1=0;}
    }
    return 0;
}
```

Fig. 3: Example program: preprocessed program

```c
#include <string.h>
#include <stdint.h>
#include <assert.h>
typedef struct {
    uint64_t gp_regs[32];
    uint64_t pc;
} cpu_t;
int assert_ret1 = 1;
extern void insertion_sort(cpu_t *c);
extern int start(cpu_t *c);
int start(cpu_t *c){
    ......
}
void insertion_sort(cpu_t *c){
    ......
}
#define STACK_SIZE 160
int main() {
    cpu_t cpu={0};
    cpu_t *c=&cpu;
    char stack[STACK_SIZE];
    char *stack_top=&stack[STACK_SIZE-1];
    c->gp_regs[2]=(uint64_t)stack_top;
    start(c);
    assert(assert_ret1 == 1);
    return 0;
}
```

Fig. 4: Example program: verifiable C code framework

avoid naming conflicts, the `main` function in the original code (Fig. 3) is renamed to `start`.

```c
void insertion_sort(int* arg0, unsigned int arg1){
    cpu_t cpu;
    cpu_t *c=&cpu;
    char stack[64];
    c->gp_regs[2]=(uint64_t)&stack[64-1];
    c->gp_regs[10]=arg0;
    c->gp_regs[11]=arg1;
L_10000: {//addi_insertion_sort
    uint64_t rs1 = c->gp_regs[2];
    c->gp_regs[2] = rs1 + (int64_t)-48LL;
    }
    ......
}
```

Fig. 5: Example program: individual function conversion

C. Binary Code Conversion

This phase is the core of our verification approach, detailed in Section V. The primary objective is to decode the ELF file into assembly code and convert it into semantically equivalent C code. Finally, for programs where verification specifications cannot be added in the high-level language, we incorporate these specifications into the converted C program using a specification generator. In the RISC-V architecture, function results are typically stored in the `x1` register, and inputs are transmitted via the `x10` to `x17` registers. Verification specifications can be manually added based on the expected input-output relationships.

Remark 1. For languages that have automated verification tools, such as C, a simple analysis to identify the interfaces of missing functions is enough to enable their isolated conversion. As demonstrated in the example shown in Fig. 3, the `insertion_sort` function can be directly converted into the program depicted in Fig. 5. This targeted conversion approach minimizes the amount of code needing conversion, thereby significantly enhancing verification efficiency.

V. Approach

In the overview section, we outlined the main steps of our verification approach. Since source code compilation and ELF file analysis are straightforward, this section focuses on converting binary programs into C programs, a key component of our approach.

After the ELF file analysis, we obtained the addresses, initial values, and names of global variables (Tag_v) and

functions (Tag_f), establishing the mapping relationships of the $\mathcal{T}$ function. We assume there are no arbitrary address read or write operations within the program, as these are typically intercepted during compilation. This means all read and write operations occur within known memory locations, specifically within the stack space or among global variables. Decoding the ELF file yields a segment of RISC-V assembly code. The core task in the binary code conversion phase is to convert this assembly code into verifiable C code. We represent the current program state as $\sigma = (Reg, Mem, FCS, pc)$. The symbols $x_{rs1}, x_{rs2} \in Reg$ denote the source registers, $x_{rd} \in Reg$ denotes the destination register, with $rs1, rs2, rd \in \{0, 1, \ldots, 31\}$. The $\texttt{offset} \in \mathbb{N}$ represents the offset and $\texttt{imm} \in \mathbb{N}$ represents the immediate number.

A. Load & Store Instruction Conversion

In RISC-V, load instructions retrieve data from memory into registers. A load instruction is expressed as

$$lx\ x_{rd},\ \texttt{offset}(x_{rs1})$$

where $lx \in \{\texttt{lb}, \texttt{lh}, \texttt{lw}, \texttt{ld}, \texttt{lbu}, \texttt{lhu}, \texttt{lwu}\}$ specifies the type of load operation.

The operational semantics of a load instruction can be expressed as:

$$\begin{aligned} &\mathcal{S}[\![lx\ x_{rd},\ \texttt{offset}(x_{rs1})]\!](\sigma) = \sigma' \\ &\sigma' = (Reg', Mem, FCS, pc + 4) \\ &Reg' = \mathcal{W}_r\left(x_{rd}, \mathcal{R}_m(\mathcal{R}_r(x_{rs1}) + \texttt{offset}, s_{lx})\right), \end{aligned} \tag{1}$$

where $s_{lx} \in Size$ is determined by the type of load instruction lx. The base address is taken from the source register x_{rs1}. By adding the $\texttt{offset}$ to this base address, we get the effective memory address. Data is then read from this memory address, with the length of the data depending on the specific load instruction used. Finally, this data is written into the destination register x_{rd}, updating the register state accordingly.

When converting load instructions to C, the first step is to check if the address is composed of the x_{rs1} register and if the offset corresponds to a global variable. For this, we present the following conversion theorem for load instructions.

Theorem 1. In RISC-V, the operational semantics of a load instruction $lx\ x_{rd}$, $\texttt{offset}(x_{rs1})$ correspond to the following ACL statement.

Case 1: $\mathcal{T}\left(\mathcal{R}_r(x_{rs1}) + \texttt{offset}\right) \in Tag_v$

$\texttt{L\_}pc\texttt{:}\{x_{rd}\texttt{=}\star\texttt{(}type\star\texttt{)(\&}x_v\texttt{);}\}$

Case 2: $\mathcal{T}\left(\mathcal{R}_r(x_{rs1}) + \texttt{offset}\right) \notin Tag_v$

$\texttt{L\_}pc\texttt{:}\{x_{rd}\texttt{=}\star\texttt{(}type\star\texttt{)(}x_{rs1}\texttt{+(}\textbf{long}\texttt{)offset);}\}$

where $type$ depends on the load instruction type.

Proof. After executing the instruction $lx\ rd$, $\texttt{offset}(x_{rs1})$, we obtain the state σ', as described in equation (1). We consider the two scenarios in the ACL program.

In case 1, where $\mathcal{T}\left(\mathcal{R}_r(x_{rs1}) + \texttt{offset}\right) \in Tag_v$, it indicates that the address $\mathcal{R}_r(x_{rs1}) + \texttt{offset}$ corresponds to a global variable x_v. In this scenario, the operational semantics of the C code transitions the program state σ to σ'' as follows:

$$\begin{aligned} &\mathcal{S}[\![\texttt{L\_}pc\texttt{:}\{x_{rd} = *(type*)(\&x_v);\}]\!](\sigma) = \sigma'' \\ &\quad \sigma'' = (Reg'', Mem, FCS, pc + 4) \\ &Reg'' = \mathcal{W}_r(x_{rd}, *(type*)(\&x_v)) \\ &\quad = \mathcal{W}_r\left(x_{rd}, R_m\left(\&x_v, \text{sizeof}(type)\right)\right) \\ &\quad = \mathcal{W}_r\left(x_{rd}, R_m\left(\mathcal{T}^{-1}(x_v), \text{sizeof}(type)\right)\right) \\ &\quad = \mathcal{W}_r\left(x_{rd}, R_m\left(\mathcal{R}_r(x_{rs1}) + \texttt{offset}, \text{sizeof}(type)\right)\right). \end{aligned}$$

In the aforementioned expression, the value of $type$ is determined by the type of the assembly instruction. For instance, the instruction lw corresponds to $type =$ int, while lwu corresponds to $type =$ unsigned int. Consequently, the size of $type$, denoted by sizeof($type$), is equivalent to s_{lx}. Thus, under this case, the states σ' and σ'' are identical.

In Case 2, the computation x_{rs1} + (long)$\texttt{offset}$ is semantically represented as $\mathcal{R}_r(x_{rs1}) + \texttt{offset}$. The casting to long ensures that the address computation is performed as a signed operation, enhancing the readability and understanding of the converted program. However, removing the cast does not impact the outcome. Thus, the semantics of this expression can be directly translated to:

$$\begin{aligned} &\mathcal{S}[\![\texttt{L\_}pc\texttt{:}\{x_{rd} = *(type*)(x_{rs1} + (\text{long})\texttt{offset});\}]\!](\sigma) \\ &=(Reg''', Mem, FCS, pc + 4) = \sigma''' \end{aligned}$$

$$Reg''' = \mathcal{W}_r\left(x_{rd}, \mathcal{R}_m\left(\mathcal{R}_r(x_{rs1}) + \texttt{offset}, \text{sizeof}(type)\right)\right)$$

Observations show that $\sigma' = \sigma'' = \sigma'''$, confirming that executing a RISC-V load instruction and its corresponding ACL program conversion results in identical program states. This verifies that the converted program maintains the same operational semantics, completing the proof. □

Next, we consider store instructions, which transfer data from a register to a memory location in RISC-V. The syntax of these instructions is expressed as follows:

$$sx\ x_{rs2},\ \texttt{offset}(x_{rs1})$$

where $sx \in \{\texttt{sb}, \texttt{sh}, \texttt{sw}, \texttt{sd}\}$ indicates the store operation, each corresponding to different data sizes. The semantic effect of these operations is defined as:

$$\begin{aligned} &\mathcal{S}[\![sx\ x_{rs2},\ \texttt{offset}(x_{rs1})]\!](\sigma) = \sigma' \\ &\sigma' = (Reg, Mem', FCS, pc + 4) \\ &Mem' = \mathcal{W}_m\left(\mathcal{R}_r(x_{rs1}) + \texttt{offset}, \mathcal{R}_r(x_{rs2}), s_{sx}\right), \end{aligned} \tag{2}$$

where $s_{sx} \in Size$ depends on the type of the sx instruction. The base address is taken from the source register x_{rs1} and added to the $\texttt{offset}$ to get the effective memory address. Data from the source register x_{rs2} is then written to this address, with the data size determined by the specific store instruction. This updates the memory state accordingly.

The conversion method for store instructions is similar to load instructions, considering whether the accessed memory address corresponds to a global variable. To address this, we present the following conversion theorem.

Theorem 2. In RISC-V, the operational semantics of a store instruction $sx\ x_{rs2}$, $\texttt{offset}(x_{rs1})$ correspond to the following ACL statement.

Case 1: $\mathcal{T}\left(\mathcal{R}_r(x_{rs1}) + \texttt{offset}\right) \in Tag_v$

$\texttt{L\_}pc\texttt{:\{*(}type*\texttt{) (\&}x_v\texttt{)=}x_{rs2}\texttt{;\}}$

Case 2: $\mathcal{T}\left(\mathcal{R}_r(x_{rs1}) + \texttt{offset}\right) \notin Tag_v$

$\texttt{L\_}pc\texttt{:\{*(}type*\texttt{) (}x_{rs1}\texttt{+(long)offset)=}x_{rs2}\texttt{;\}}$

where $type$ depends on the store instruction type.

The proof of Theorem 2 follows a similar logic and structure to that of Theorem 1. Therefore, to avoid redundancy, we will not elaborate further on the proof.

B. Jump Instruction Conversion

Jump instructions in RISC-V present significant challenges, particularly in identifying return addresses without dynamic function execution. This difficulty stems from the inability to determine where a function will resume after completion using static analysis alone. To address this complexity, our analysis initially focuses on the direct jump instruction `jal`.

The `jal` performs a direct unconditional jump to a calculated address, simultaneously saving the address of the following instruction into a designated link register. The syntax and operational semantics of `jal` are given by

$$\texttt{jal}\ x_{rd}, \texttt{offset}$$

In this instruction, the target address for the jump is computed by adding the `offset` to the pc, and the return address (pc + 4) is stored in x_{rd}. The operational semantics of `jal` can be formally defined as:

$$\begin{aligned}&\mathcal{S}[\![\texttt{jal}\ x_{rd}, \texttt{offset}]\!](\sigma) = (Reg', Mem, FCS, pc') = \sigma',\\ &Reg' = \mathcal{W}_r\left(x_{rd}, pc + 4\right),\\ &pc' = pc + \texttt{offset}.\end{aligned} \tag{3}$$

In RISC-V, the `jal` instruction, typically used as a function call (`call` pseudo-instruction), stores the return address in x_1. In this case, the `jal` instruction becomes:

$$\texttt{jal}\ x_1, \texttt{offset}$$

When a function call occurs, the FCS within the program state changes accordingly. It automatically pushes the return address of the current function onto the stack. Therefore, the semantics of the `call` pseudo-instruction are described as follows:

$$\begin{aligned}&\sigma'' = (Reg'', Mem, push\left(\mathcal{R}_r\left(x_{rd}\right)\right), pc''),\\ &Reg'' = Reg', \quad pc'' = pc'.\end{aligned} \tag{4}$$

To address the conversion of the `jal` instruction, we propose the following theorem.

Theorem 3. In RISC-V, the operational semantics of a load instruction $\texttt{jal}\ x_{rd}, \texttt{offset}$ correspond to the following ACL statement.

Case 1: for the `call` pseudo-instruction

$\texttt{L\_}pc\texttt{:\{}x_{rd}\texttt{=}pc + 4\texttt{;f();\}}$

Case 2: for general use

$\texttt{L\_}pc\texttt{:\{}x_{rd}\texttt{=}pc + 4\texttt{;}\textbf{goto}\ \texttt{L\_(}pc\texttt{+offset);\}}$

Proof. Firstly, we analyze the scenario where the `jal` instruction functions as a `call` pseudo-instruction. In this case, $\mathcal{T}(pc\texttt{+offset}) = f \in Tag_f$ confirms that the target address of the jump is recognized as a function address within the set of function tags.

$$\begin{aligned}&\mathcal{S}[\![\texttt{L\_}pc\texttt{:}\{x_{rd} = pc + 4; f();\}]\!](\sigma)\\ =&\mathcal{S}[\![f();]\!]\left(\mathcal{S}[\![x_{rd} = pc + 4;]\!](\sigma)\right)\\ =&\mathcal{S}[\![f();]\!]\left(\mathcal{W}_r(x_{rd}, pc + 4), Mem, FCS, pc\right)\\ =&\left(\mathcal{W}_r(x_{rd}, pc + 4), Mem, push(pc + 4), \mathcal{T}^{-1}(f)\right)\\ =&\sigma'''\end{aligned}$$

Analyzing the above formulation reveals that σ''' equals σ'' as defined in equation (4). Now, let's consider the second case.

$$\begin{aligned}&\mathcal{S}[\![\texttt{L\_}pc\texttt{:}\{x_{rd} = pc + 4; \texttt{goto L\_(}pc + \texttt{offset);}\}]\!](\sigma)\\ =&\mathcal{S}[\![\texttt{goto L\_(}pc + \texttt{offset);}]\!]\left(\mathcal{S}[\![x_{rd} = pc + 4;]\!](\sigma)\right)\\ =&\left(\mathcal{W}_r(x_{rd}, pc + 4), Mem, FCS, pc + \texttt{offset}\right)\\ =&\sigma''''\end{aligned}$$

By analyzing the above expression and comparing it with equation (4) , we can see that $\sigma'''' = \sigma''$. Hence, Theorem 3 is proved. □

Next, we analyze the `jalr` instruction. For general cases, the `jalr` instruction in RISC-V typically functions as a dynamic jump, calculating the target address from a base register and an offset. It is defined as:

$$\texttt{jalr}\ x_{rd}, \texttt{offset}(x_{rs1})$$

The `jalr` instruction computes the target address by adding the `offset` to the value in the base register x_{rs1} and then jumps to this computed address, storing the return address in x_{rd}. The operational semantics of `jalr` is given by

$$\begin{aligned}&\mathcal{S}[\![\texttt{jalr}\ x_{rd}, \texttt{offset}(x_{rs1})]\!](\sigma)\\ =&(Reg''', Mem, FCS, \mathcal{R}_r(x_{rs1}) + \texttt{offset}),\\ &Reg''' = \mathcal{W}_r(x_{rd}, pc + 4).\end{aligned} \tag{5}$$

When `jalr` is used as the `ret` pseudo-instruction, it functions to return from a function by jumping to the address stored in register x_1, typically used to store the return address. The instruction format simplifies to:

$$\texttt{jalr}\ x_0, 0(x_1)$$

where x_0 is always zero (as writes to x_0 have no effect due to it being a hardwired zero), x_1 holds the return address, and the offset is zero. This setup utilizes the address stored in x_1 directly from the FCS. The semantics for the `ret` pseudo-instruction is thus expressed as:

$$\mathcal{S}[\![\texttt{ret}]\!](\sigma) = (Reg, Mem, pop(FCS), top(FCS)) \tag{6}$$

Theorem 4. In RISC-V, the operational semantics of a load instruction `jalr` x_{rd}, `offset`(x_{rs1}) correspond to the following ACL statement.

Case 1: for the `ret` pseudo-instruction

L_pc:{**return;**}

Case 2: for general `jalr` usage

L_pc:{x_{rd}=pc+4;**goto** L_(x_{rs1}+offset);}

Proof. For csase 1, referencing Definition 4 regarding the `return` statement, we establish:

$$\mathcal{S}[\![\texttt{return;}]\!](\sigma) = (Reg, Mem, pop(FCS), top(FCS))$$

This resulting state is identical to that described in equation (6). For case 2, the analysis parallels that of case 2 in Theorem 3. Consequently, this completes the proof of the theorem. □

C. Conditional Branch Instruction Conversion

In RISC-V, conditional branch instructions direct the flow of execution based on the comparison of two registers. A conditional branch instruction is expressed as:

$$bx\ x_{rs1},\ x_{rs2},\ \texttt{offset}$$

where $bx \in \{\texttt{beq}, \texttt{bne}, \texttt{blt}, \texttt{bge}, \texttt{bltu}, \texttt{bgeu}\}$ specifies the type of branch operations. These operations compare the values in registers x_{rs1} and x_{rs2}. If the condition specified by bx is satisfied, the program counter is updated to the address obtained by adding the `offset` to the value in x_{rs1}. The operational semantics of a conditional branch instruction can be described as:

$$\mathcal{S}[\![bx\ x_{rs1}, x_{rs2}, \texttt{offset}]\!](\sigma) = \sigma'$$

$$\sigma' = \begin{cases} (Reg, Mem, FCS, pc + \texttt{offset}) & \text{if condition holds} \\ (Reg, Mem, FCS, pc + 4) & \text{otherwise} \end{cases} \tag{7}$$

where the condition represents whether the expression $\mathcal{R}_r(x_{rs1})\ \texttt{op}\ \mathcal{R}_r(x_{rs2})$ holds, where `op` can be any of the operators ==, !=, <, >=.

Theorem 5. In RISC-V, the operational semantics of a load instruction $bx\ x_{rs1}$, x_{rs2}, `offset` correspond to the following ACL statement.

L_pc:{**if**(con)**goto** L_(pc+offset);}

where the con denotes the condition expression $(type)x_{rs1}\ \texttt{op}\ (type)x_{rs2}$, the selection of $type$ (either `long` or `unsigned long`) depends entirely on the specific branch instruction being converted, determining how x_{rs1} and x_{rs2} are compared within the condition.

Proof. Analyzing the semantics of the converted C command yields the following insights:

$$\mathcal{S}[\![\texttt{L\_}pc\texttt{:\{if(con)goto L\_(}pc\texttt{+offset);\}}]\!](\sigma) = \sigma''$$

$$\sigma'' = \begin{cases} \mathcal{S}[\![\texttt{goto L\_(}pc\texttt{+offset);}]\!](\sigma) & con \neq 0 \\ (Reg, Mem, FCS, pc + 4) & \text{otherwise} \end{cases}$$

$$= \begin{cases} (Reg, Mem, FCS, pc + \texttt{offset}) & con \neq 0 \\ (Reg, Mem, FCS, pc + 4) & \text{otherwise} \end{cases}$$

Upon analyzing the condition `con` alongside the equation, it is evident that $\sigma' = \sigma''$. Therefore, this completes the proof of the theorem. □

D. Immediate & Computation Instructions Conversion

The conversion of immediate and computation instructions within the RISC-V architecture is relatively straightforward. This subsection briefly introduces the conversion methods for these instructions, focusing on immediate instructions initially.

Immediate instructions, such as `lui` and `auipc`, primarily aim to write an immediate value into a target register. We propose the following conversion theorem for these instructions.

Theorem 6. In RISC-V, the immediate instructions `lui` x_{rd}, `imm` and `auipc` x_{rd}, `imm` convert in ACL statements to `lui`: $\{x_{rd} = \texttt{imm} << 12;\}$ and `auipc`: $\{x_{rd} = pc + (\texttt{imm} << 12);\}$.

The operational semantics of RISC-V computation instructions are divided into register-based (`calc` x_{rd}, x_{rs1}, x_{rs2}) and immediate-based (`calc` x_{rd}, x_{rs1}, `imm`) types. Here, `calc` refers to computational operations such as addition, multiplication, or logical shifts. These instructions perform calculations between two source registers (x_{rs1} and x_{rs2}), or a source register and an immediate value (x_{rs1} and `imm`), with results stored in the destination register (x_{rd}). Due to their straightforward nature, conversions to pseudocode are presented directly.

Theorem 7. In RISC-V, the register-based computation instruction `calc` x_{rd}, x_{rs1}, x_{rs2} converts to an ACL statement as $\{x_{rd} = x_{rs1}\ op\ x_{rs2};\}$, and the immediate-based computation instruction `calc` x_{rd}, x_{rs1}, `imm` converts to $\{x_{rd} = x_{rs1}\ op\ \texttt{imm};\}$, where op represents a binary operation determined by the instruction type.

E. Summary

Based on the analyses presented in the previous subsections, this paper has detailed the conversion methodologies for various instructions, excluding simple arithmetic and immediate instructions, and has demonstrated the semantic equivalence between the converted C programs and the original assembly instructions. It is established that starting from any program state σ, the state σ' reached by executing any assembly instruction listed in Table 1 is equivalent to the state obtained by executing its corresponding converted ACL code. From straightforward inductive reasoning, we can derive the following theorem based on these conversions.

Theorem 8. Assuming the binary program compiled with the official compiler is operationally semantically equivalent to the original program, then for any RISC-V program consisting only of instructions from Table I, the C program obtained by applying the conversion rules from Theorems 1 to 7 is operationally semantically equivalent to the original program.

This section detailed the conversion methods for common RISC-V instructions and provided theoretical proof of semantic equivalence between the heterogeneous program and its C

TABLE II: Assembly Verification Results Comparison

Metric	Correct Assembly			Incorrect Assembly		
	Unwind 20	Unwind 50	Unwind 100	Unwind 50	Unwind 100	No Limit
Correct Count	50	34	29	38	43	48
Timeout Count	0	16	21	2	2	2
Correct Rate	100%	68%	58%	76%	86%	96%
Error Rate	0%	0%	0%	20%	10%	0%

Note: The "unwind" parameter in CBMC specifies the number of loop iterations explored during verification.

TABLE III: Verification Results of Large Heterogeneous Programs

Program Type	Program Count	Verification Approach	Metrics			
			Correct Count	Timeout Count	Correct Rate	Error Rate
Seq-mthreaded	6	Direct	0	0	0%	100%
		Post-conversion	**6**	**0**	**100%**	**0%**
Combinations	11	Direct	3	8	27%	73%
		Post-conversion	**11**	**0**	**100%**	**0%**
Loops	1	Direct	1	0	100%	0%
		Post-conversion	**1**	**0**	**100%**	**0%**
Product-lines	7	Direct	4	0	57%	43%
		Post-conversion	**7**	**0**	**100%**	**0%**
Systems	5	Direct	3	0	60%	40%
		Post-conversion	**5**	**0**	**100%**	**0%**

conversion. Additionally, based on the discussed theorems, we established the operational equivalence between the original program and the converted C code.

VI. Evaluation

This section evaluates the effectiveness and advantages of the Hetrify tool through three key experiments, focusing on its performance in verifying assembly, large heterogeneous, and binary programs. In order to comprehensively evaluate Hetrify's performance, we investigated the following research questions:

- **RQ1:** Can Hetrify effectively verify low-level language programs?
- **RQ2:** Is Hetrify capable of verifying large-scale heterogeneous programs with missing source code functions?
- **RQ3:** Can Hetrify be replaced by existing decompiler tools such as IDA Pro and Ghidra?

The experiments were conducted using CBMC (version 5.95) on a system equipped with dual AMD EPYC 7713 CPUs, two A800 GPUs, and 384 GB of RAM, running Ubuntu 22.04 LTS, and the verification results were assessed using two key metrics:

$$\text{Correctness Rate} = \frac{\text{Correct Verifications}}{\text{Total Tests}},$$

$$\text{Error Rate} = 1 - \frac{\text{Correct Verifications} + \text{Timeouts}}{\text{Total Tests}}.$$

A. *RQ1: Can Hetrify effectively verify low-level language programs?*

To answer RQ1, this experiment was designed to evaluate Hetrify's effectiveness in verifying low-level language programs, specifically assembly, while also validating the accuracy of our proposed assembly-to-C conversion methodology. To ensure the experiment's rigor and professionalism, We randomly selected 100 C programs from SV-COMP 2024 [27], equally split between 50 correct and 50 incorrect programs, each containing one to three functions. To facilitate compatibility with the Hetrify tool and the RISC-V compiler, we modified these C programs by eliminating certain non-essential functions, ensuring that these alterations did not affect the verification outcomes. We then compiled these adjusted programs into assembly code using the `riscv64-unknown-elf-gcc` compiler to create our targeted assembly program dataset.

We employed the Hetrify tool to convert these 100 assembly programs into verifiable C programs. Subsequently, we utilized CBMC to individually verify each of the converted C programs, setting a verification time limit of 3000 seconds per program. The results are presented in Table II.

The results in Table II demonstrate Hetrify's effectiveness in verifying low-level language (RISC-V assembly) programs. For correct programs, a 100% success rate was achieved at an unwind factor of 20, but it decreased with higher unwind factors due to increased computational complexity. In contrast, for incorrect programs, the success rate improved as the unwind factor increased, reaching 96% with no unwind limit. These findings suggest that, theoretically, given sufficient time, Hetrify could successfully verify all low-level programs without encountering verification errors.

TABLE IV: Binary Program Verification Results Comparison

Metric	Correct Program			Incorrect Program		
	IDA Pro	Ghidra	Hetrify	IDA Pro	Ghidra	Hetrify
Convertible Count	0	50	**50**	0	47	**50**
Correct Count	-	36	**29**	-	47	**48**
Timeout Count	-	6	**21**	-	0	**2**
Correct Rate	0%	72%	**58%**	0%	94%	**96%**
Error Rate	-	16%	**0%**	-	0%	**0%**

Note: The correct program is loop-unwound up to 100 times, while the incorrect program has no unwinding limit.

B. ***RQ2:*** *Is Hetrify capable of verifying large-scale heterogeneous programs with missing source code functions?*

To verify this question, we conducted an experiment to assess Hetrify's ability to verify large-scale heterogeneous programs, particularly when source code for certain functions is missing. Due to the limited number of qualifying programs in SV-COMP 2024 [27], additional programs were sourced from SV-COMP 2023 [28]. A total of 30 verification programs were selected, comprising 15 correct and 15 incorrect programs. Each program contains more than 5 functions and exceeds 300 lines of code, with an average length of over 1000 lines. CBMC successfully verified all selected programs.

We randomly extracted functions highly correlated with the verification results from these 30 programs and compiled them into a static library using the `riscv64-unknown-elf-gcc` compiler, subsequently removing these functions from the source files. Following this, Hetrify was used to convert these static library files, and verification was conducted in conjunction with the source files. Additionally, a comparative experiment was also set up for direct verification of the source files with the missing functions. The timeout was set to 3000 seconds, with a fixed unwind factor of 20 for correct programs and no fixed unwind factor for incorrect programs. The verification results are shown in Table III.

The results in Table III show that the total correctness rate for direct verification across all five program types was 36.7% with an error rate of 63.3%. In contrast, post-conversion verification achieved a perfect correctness rate of 100% with no errors. This demonstrates that post-conversion verification significantly outperforms direct verification in all program types. Hetrify effectively addresses the verification challenges posed by large-scale heterogeneous programs, enabling accurate verification even when source code for certain functions is missing.

C. ***RQ3:*** *Can Hetrify be replaced by existing decompiler tools such as IDA Pro and Ghidra?*

To verify this question, we experimented to determine whether traditional decompilers, such as IDA Pro and Ghidra, could serve as viable alternatives to Hetrify. We selected 100 assembly programs, evenly split between 50 correct and 50 incorrect ones, and compiled them into binaries. Each binary was then decompiled using IDA Pro, Ghidra, and Hetrify, and the resulting C code was verified using CBMC.

During the experiment, we encountered several limitations with traditional decompilers. IDA Pro could not process RISC-V binaries at all, making it unsuitable for this verification task. Ghidra, although capable of decompiling the binaries, generated C code with frequent syntax issues, such as missing header file declarations, undefined global variables, and incorrect variable types. These issues required manual corrections, which we applied before attempting verification. However, despite these corrections, the generated C code still likely contained semantic inaccuracies.

The verification results are summarized in Table. IV. As shown, Ghidra encountered issues during the conversion process, with 3 programs failing due to unresolvable syntax errors caused by incorrect function interface conversions. Additionally, 8 programs that were successfully converted still resulted in verification errors. In contrast, Hetrify produced correct and verifiable C code without requiring any manual corrections, demonstrating its clear advantage over traditional decompilers. These results suggest that Hetrify offers significant benefits and cannot be replaced by existing decompilers like IDA Pro and Ghidra, particularly when dealing with RISC-V binaries.

VII. RELATED WORK

Assembly to High-Level Languages. Converting assembly to high-level languages has seen significant advancements through both AI-based methods and traditional techniques. SLaDe employs a transformer model to improve the accuracy and readability of assembly-to-C translations [29]. Neutron achieves over 97% accuracy in translating low-level languages using an attention-based model [30]. TraFix generates training data and employs verifiers to ensure high success rates in decompiling LLVM IR and x86 assembly to C [31]. Research on GPT-4 demonstrates its potential in code understanding, though it has limitations in detailed analysis [12]. Other AI-driven methods [11], [32]–[34], alongside contributions from projects like LLM4Decompile [35] and Codex [12], have significantly enhanced the translation process, although challenges in ensuring semantic equivalence remain.

Traditional assembly-to-C translation tools, such as Relogix [36], Boomerang [37] , Hex-Rays decompiler [10], and Radare2 [38], remain essential. Relogix is praised for converting complex assembly instructions into maintainable C code, though it can be costly. Boomerang excels in reverse engineering but may not always produce perfect results. Hex-Rays, integrated with IDA Pro, excels in reverse engineering

but isn't ideal for automatic conversions. Radare2 offers an extensive toolset but faces challenges due to the complex nature of assembly code. Despite advancements in AI methods, these traditional tools remain invaluable in the field.

Assembly Code-Based Verification. Various methodologies have emerged for assembly code-based verification, each with distinct advantages and challenges [15], [39]–[42]. Fehnker et al. use model-checking for ARM inline assembly, potentially missing deeper semantic details [43]. Maus's technique involves simulating assembly instructions in a virtual machine, providing detailed verification but increasing complexity [17]. Myreen et al. focus on verifying pure assembly code through labor-intensive semantic equivalence proofs [44]. SMT-based methods improve accuracy for embedded programs but face performance issues with large-scale code [45]. CompCertM extends the CompCert compiler for multi-language linking using the RUSC technique, though scalability remains a concern [46]. These studies illustrate both the advancements and ongoing challenges in achieving reliable assembly code verification.

Program Semantic Equivalence Verification. In the domain of program semantic equivalence verification, various methods have been developed, each with unique strengths and limitations. [47] uses the K framework to formalize MIPS assembly language, enabling modular verification but increasing complexity for intricate code. The Vale tool converts annotated assembly to an abstract syntax tree and uses SMT solvers for verification, excelling in high-performance cryptographic code verification despite its complexity [42]. [48] introduces a relational correctness proof method for static analysis and program transformation, which, while theoretically robust, may face scalability and performance issues in large-scale applications. [49] automates the verification of functional programming assignments, showing strong performance with many submissions but facing challenges with complex recursion. These studies highlight significant advancements and ongoing challenges in achieving comprehensive program semantic equivalence verification.

VIII. Discussion

This section discusses the impact of the proposed conversion approach on performance and scalability, focusing on the increase in lines of code and the comparison of verification times between the original and lifted C code. We analyze these factors to understand better the trade-offs and benefits of adopting this approach.

A. Increase in Lines of Code

Our approach involves converting each assembly instruction into one or two lines of C code. As a result, the length of the converted code directly depends on the length of the original assembly code. This method ensures that the increase in lines of code is proportional to the size of the assembly code being converted. Furthermore, Hetrify's conversion speed is exceptionally fast, with the entire conversion process completed almost instantly, allowing for efficient code conversion without significant delays.

B. Verification Time Comparison

This work's main contribution is making the verification of low-level programs, such as binaries and assembly code, possible—something that is not supported by existing verification tools. For the programs in Experiment 1, after being compiled into assembly and then converted to C code using Hetrify, the resulting C code is significantly larger compared to the original SV-COMP C code. This is due to the fact that compiling C code typically generates several times more assembly instructions. After conversion, the program's logical structure becomes more complex, leading to a significant increase in verification time. Therefore, reducing the overhead of verification time will be a key focus of our future research.

IX. Conclusion

This paper addresses the issue of inaccurate verification results for heterogeneous programs. We propose a novel verification approach and introduce a RISC-V-based tool named Hetrify. The core idea is to convert unverifiable segments or entire heterogeneous programs into verifiable C code and then verify these converted C programs alongside the original code. To ensure semantic equivalence between the converted C programs and the original programs, we provide detailed mathematical proofs. Experiments on assembly and heterogeneous program verification demonstrate the effectiveness of our approach and the accuracy of our mathematical proofs.

X. Data Availability

Hetrify and its experimental data are publicly available at https://anonymous.4open.science/r/Hetrify

Acknowledgments

This work was funded by the National Nature Science Foundation of China (No. U2341212, No. 62032024); and the National key R&D program of China (No. 2022YFB4501903).

References

[1] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[2] K. Crowston and J. Howison, "The social structure of free and open source software development," 2005.

[3] P. Devanbu, T. Zimmermann, and C. Bird, "Belief & evidence in empirical software engineering," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 108–119. [Online]. Available: https://doi.org/10.1145/2884781.2884812

[4] I. Sommerville, "Software engineering (ed.)," *America: Pearson Education Inc*, 2011.

[5] B. Chess and J. West, *Secure programming with static analysis*. Pearson Education, 2007.

[6] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001, pp. 203–213.

[7] T. Ball and S. K. Rajamani, "The SLAM project: Debugging system software via static analysis," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002, pp. 1–3.

[8] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *International Conference on Compiler Construction*. Springer, 2002, pp. 213–228.

[9] Zynamics GmbH, "Bindiff," https://www.zynamics.com/bindiff.html.

[10] Hex-Rays SA, "Hex-Rays decompiler," https://www.hex-rays.com/products/decompiler/.

[11] I. Hosseini and B. Dolan-Gavitt, "Beyond the C: Retargetable decompilation using neural machine translation," in *Proceedings 2022 Workshop on Binary Analysis Research*, ser. BAR 2022. Internet Society, 2022. [Online]. Available: http://dx.doi.org/10.14722/bar.2022.23009

[12] S. Pordanesh and B. Tan, "Exploring the efficacy of large language models (GPT-4) in binary reverse engineering," 2024. [Online]. Available: https://arxiv.org/abs/2406.06637

[13] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 472–489.

[14] S. Schmaltz and A. Shadrin, "Integrated semantics of intermediate-language C and macro-assembler for pervasive formal verification of operating systems and hypervisors from VerisoftXT," in *International Conference on Verified Software: Tools, Theories, Experiments*. Springer, 2012, pp. 18–33.

[15] Frédéric, S. Bardin, R. Bonichon, L. Mounier, and M.-L. Potet, "Get rid of inline assembly through verification-oriented lifting," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 577–589.

[16] ksco, "rvemu: RISC-V emulator," https://github.com/ksco/rvemu.git, 2024.

[17] S. Maus, "Verification of hypervisor subroutines written in assembler," Ph.D. dissertation, PhD thesis, Freiburg University, Computer Science Department, 2011.

[18] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54*, p. 4, 2014.

[19] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual volume II: Privileged architecture version 1.7," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-49*, 2015.

[20] P. Pieper, V. Herdt, and R. Drechsler, "Advanced embedded system modeling and simulation in an open source RISC-V virtual prototype," *Journal of Low Power Electronics and Applications*, vol. 12, no. 4, p. 52, 2022.

[21] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[22] D. Kroening and M. Tautschnig, "CBMC–C bounded model checker: (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*. Springer, 2014, pp. 389–391.

[23] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," in *Model Checking Software: 13th International SPIN Workshop, Vienna, Austria, March 30-April 1, 2006. Proceedings 13*. Springer, 2006, pp. 146–162.

[24] D. Beyer, "Competition on software verification and witness validation: SV-COMP 2023," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2023, pp. 495–522.

[25] D. Beyer, "State of the art in software verification and witness validation: SV-COMP 2024," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2024, pp. 299–329.

[26] F. Nielson and H. R. Nielson, *Formal Methods*. Springer, 2019.

[27] "SV-COMP 2024," https://sv-comp.sosy-lab.org/2024/results/results-verified/META_ReachSafety.table.html#/table?hidden=0,1,2,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25&filter=3(0*status*(category(in(correct)))), 2024.

[28] "SV-COMP 2023," https://sv-comp.sosy-lab.org/2023/results/results-verified/META_ReachSafety.table.html#/table?hidden=0,1,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23&filter=2(0*status*(category(in(correct)))), 2023.

[29] J. Armengol-Estapé, J. Woodruff, C. Cummins, and M. F. O'Boyle, "SLaDe: A portable small language model decompiler for optimized assembly," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024, pp. 67–80.

[30] R. Liang, Y. Cao, P. Hu, and K. Chen, "Neutron: an attention-based neural decompiler," *Cybersecurity*, vol. 4, pp. 1–13, 2021.

[31] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, "Towards neural decompilation," 2019. [Online]. Available: https://arxiv.org/abs/1905.08325

[32] X. She, Y. Zhao, and H. Wang, "WaDec: Decompile webassembly using large language model," 2024. [Online]. Available: https://arxiv.org/abs/2406.11346

[33] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," 2020. [Online]. Available: https://arxiv.org/abs/2006.03511

[34] Y. Huang, M. Qi, Y. Yao, M. Wang, B. Gu, C. Clement, and N. Sundaresan, "Program translation via code distillation," 2023. [Online]. Available: https://arxiv.org/abs/2310.11476

[35] H. Tan, Q. Luo, J. Li, and Y. Zhang, "LLM4Decompile: Decompiling binary code with large language models," 2024. [Online]. Available: https://arxiv.org/abs/2403.05286

[36] MicroAPL Ltd, "Relogix Assembler-to-C translator," http://www.microapl.co.uk/asm2c/index.html, accessed: date-of-access.

[37] Boomerang Decompiler Project, "Boomerang open source decompiler," http://boomerang.sourceforge.net.

[38] Radare2 Team, "Radare2: A portable reversing framework," https://www.radare.org/.

[39] F. Verbeek, J. A. Bockenek, and B. Ravindran, "Highly automated formal proofs over memory usage of assembly code," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Biere and D. Parker, Eds. Cham: Springer International Publishing, 2020, pp. 98–117.

[40] D. W. Currie, "A tool for formal verification of DSP assembly language programs," Ph.D. dissertation, University of British Columbia, 1999. [Online]. Available: https://open.library.ubc.ca/collections/ubctheses/831/items/1.0051611

[41] J.-P. Zha, X.-Y. Feng, and L. Qiao, "Modular verification of SPARCv8 code," *Journal of Computer Science and Technology*, vol. 35, no. 6, pp. 1382–1405, 2020.

[42] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance Cryptographic assembly code," in *26th USENIX security symposium (USENIX security 17)*, 2017, pp. 917–934.

[43] A. Fehnker, R. Huuck, F. Rauch, and S. Seefried, "Some assembly required-program analysis of embedded system code," in *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2008, pp. 15–24.

[44] M. O. Myreen, M. J. Gordon, and K. Slind, "Machine-code verification for multiple architectures-an application of decompilation into logic," in *2008 Formal Methods in Computer-Aided Design*. IEEE, 2008, pp. 1–8.

[45] S. Yamane, J. Kobashi, and K. Uemura, "Verification method of safety properties of embedded assembly program by combining SMT-Based bounded model checking and reduction of interrupt handler executions," *Electronics*, vol. 9, no. 7, 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/7/1060

[46] Y. Song, M. Cho, D. Kim, Y. Kim, J. Kang, and C.-K. Hur, "CompCertM: Compcert with C-assembly linking and lightweight modular verification," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–31, 2019.

[47] M. Asăvoae, "K semantics for assembly languages: A case study," *Electronic Notes in Theoretical Computer Science*, vol. 304, pp. 111–125, 2014.

[48] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 14–25, 2004.

[49] D. Milovančević and V. Kunčak, "Proving and disproving equivalence of functional programming assignments," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 928–951, 2023.