No More Translation at Runtime: LLM-Empowered Static Binary Translation

Zhibo Liu*
State Key Laboratory of Novel Software Technology, Nanjing University
Nanjing, China
zhiboliu@nju.edu.cn

Huaijin Wang
The Hong Kong University of Science and Technology
Hong Kong, China
hwangdz@cse.ust.hk

Wai Kin Wong
The Hong Kong University of Science and Technology
Hong Kong, China
wkwongal@cse.ust.hk

Daoyuan Wu*
Lingnan University
Hong Kong, China
daoyuanwu@ln.edu.hk

Shuai Wang†
The Hong Kong University of Science and Technology
Hong Kong, China
shuaiw@cse.ust.hk

Abstract

While AArch64 CPUs are becoming strong market contenders, their software ecosystem lags behind the mature x86-64 environment, hindering the adoption of the new architectures and impacting user experience. *Binary translation* bridges this divide by converting binary code from one architecture (e.g., x86-64) to run on another (e.g., AArch64), allowing legacy software to benefit from modern hardware's performance and energy efficiency advantages.

Current translation methods are typically either dynamic, which adds significant runtime overhead, or static, which struggles with reliability due to the inherent complexities of binary analysis. This paper introduces a new static, assembly-to-assembly translation paradigm that transforms binary code ahead of execution, generating portable, efficient native-like binaries that run on AArch64 devices without runtime frameworks. Benefiting from recent breakthroughs in large language models (LLMs), we provide a practical and automated translation engine that produces high-quality code with minimal human intervention. To ensure correctness, we introduce a crucial verification step, where we split the assembly code into simplified snippets, enabling efficient and scalable semantic verification.

Our evaluation shows that this approach significantly outperforms existing open-source solutions with a large margin, producing binaries with near-native performance. Furthermore, it shows substantial improvements over the leading industrial translator, ExaGear, illuminating a promising new direction for cross-architecture binary translation research.

CCS Concepts: • **Software and its engineering** → **Translator writing systems and compiler generators**; *Automatic programming*; *Software verification*; • **Security and privacy** → *Software reverse engineering*; • **Computer systems organization** → *Architectures.*

Keywords: Binary Translation, Cross-Architecture Migration, Large Language Models, Reverse Engineering

ACM Reference Format:
Zhibo Liu, Huaijin Wang, Wai Kin Wong, Daoyuan Wu, Shuai Wang. 2026. No More Translation at Runtime: LLM-Empowered Static Binary Translation. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3767295.3803600

1 Introduction

The computing hardware landscape is shifting away from the long-standing dominance of x86-64, driven by the rise of alternative Instruction Set Architectures (ISAs) like ARM and RISC-V [43, 93]. AArch64 has already achieved significant success in mobile devices and is now making remarkable strides in the PC market with processors like Apple's Silicon chips [5, 6]. Their high performance and energy efficiency have spurred major manufacturers to develop and promote AArch64 CPUs. Notable examples include Huawei's Kunpeng server chips [54] and Qualcomm's Snapdragon X Elite processor for Windows laptops [80, 89].

Despite the advantages of ARM-based chips, their widespread adoption is hindered by a major software hurdle. While the AArch64 software ecosystem is thriving, decades of software development have been optimized for x86-64.

*Work done while the author was associated with HKUST.
†Corresponding author.

EUROSYS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3803600

Legacy applications often lack available source code or complete build environments, making a source-based recompilation non-viable [7].

To bridge this gap, chip manufacturers and researchers have turned to binary translation[8, 17, 80], which converts binary code from one ISA to another, allowing seamless software migration from an old architecture to a new one. Prominent examples include Apple's Rosetta 2 [8] for macOS, Huawei's ExaGear [58] for its Kunpeng-powered Linux systems, and Microsoft's Prism [80] for Windows on ARM.

While existing binary translation solutions have facilitated the rapid growth of the AArch64 ecosystem, they are often limited by their low reliability or high runtime overhead. Specifically, static translators lift assembly code to a higher-level, platform-independent intermediate representation (IR) (e.g., LLVM IR) before recompiling it for a new platform [12, 114]. This process is fundamentally challenged by the undecidable problem of statically recovering high-level semantic information [55], such as symbols and variables, that is discarded during compilation. Differently, dynamic solutions translate code at runtime [15, 41, 58], which not only incurs extra performance overhead, but also restricts the portability of translated software, preventing it from migrating to platforms without the translation framework.

Moreover, state-of-the-art binary translation tools are closed-source and platform or CPU-specific [8, 58, 80], creating barriers for research aiming to enhance modern binary translation techniques. The complexity of modern ISAs makes developing a cross-architecture translator a daunting task, requiring substantial expertise and effort to identify equivalent instructions across ISAs and examine their subtle semantic differences. These challenges have slowed academic research, highlighting the need for an automated solution.

To address these limitations, we present a new paradigm for cross-architecture binary translation. Leveraging recent advances in reverse engineering research and LLMs, we propose a *static assembly-to-assembly* translation paradigm that converts x86-64 binaries to AArch64 *ahead of time*, eliminating runtime translation overhead while producing portable binaries that can run natively on AArch64 platforms. Furthermore, given LLMs' success in multilingual translation and various coding tasks [21, 26, 48, 67, 72], we explore their potential to facilitate binary translator development, minimizing the reliance on extensive expert knowledge. Additionally, we parameterize translation results to generate reusable translation rules to expedite future translations.

In summary, this paper makes the following contributions:

- We advocate for increased research in binary translation to support software migration for new architectures like AArch64. To this end, we introduce a novel static assembly-to-assembly translation paradigm that utilizes advanced binary rewriting techniques to produce portable binaries with performance comparable to native counterparts.
- We present an automatic binary translation pipeline that leverages LLMs to construct the translation engine, drastically reducing the reliance on costly expert knowledge and manual effort. Meanwhile, our approach simplifies the translation task for LLMs by splitting code into snippets, allowing for straightforward equivalence checking to ensure correctness and scalability.
- We present an academic binary translation framework with performance that rivals leading industrial tools. Evaluation shows that our research prototype, LLMSbt, significantly outperforms existing open-source solutions and even surpasses a leading industrial dynamic binary translator, ExaGear, in some real-world scenarios. The translated binaries achieve near-native performance while remaining portable. We also analyze the remaining runtime overheads, discuss LLMSbt's limitations, and suggest potential improvements for future research.

2 Background

2.1 Binary Analysis and Reverse Engineering

Disassembly. Disassembly is the foundational process of converting binary into human-readable assembly. Two primary disassembly techniques are linear sweeping and recursive traversal. Linear sweeping [33] decodes all bytes in the executable region sequentially, while recursive traversal [65] begins at a designated entry point and decodes following control transfer instructions. Recently, machine learning techniques have emerged, utilizing neural networks to map binary bytes to assembly instructions [84, 115, 116]. Despite decades of research, perfect disassembly remains elusive. Disassemblers struggle with indirect jumps and calls, as they cannot statically determine all control flow targets, often resulting in incomplete and inaccurate outputs [4, 64].

Symbolization. A key challenge in reverse engineering is the loss of symbolic information during compilation. Programmer defined symbols such as functions, variables, and labels are discarded, leaving only constant offsets in the binary. As depicted in Fig. 1, these symbols become indistinguishable from other constants, leading to what is known as the *symbolization problem* [103, 104]. This poses a significant hurdle for recompilation, as new addresses assigned by compilers may lead to incorrect symbol references.

To address this, researchers have developed advanced *reassemblable disassembling* techniques [38, 42, 103, 104, 110]. Notable approaches leverage binary analysis and relocation information in position-independent code (PIC) to identify code and data pointers [38, 110]. They focus on recovering lost symbols in disassembled code and have proven highly reliable. Recent advancements have even introduced sound binary rewriting techniques [63], with many resulting tools rigorously tested and validated [64] in numerous real-world applications. Fig. 1 illustrates how proper symbolization enables correct symbol references during recompilation.

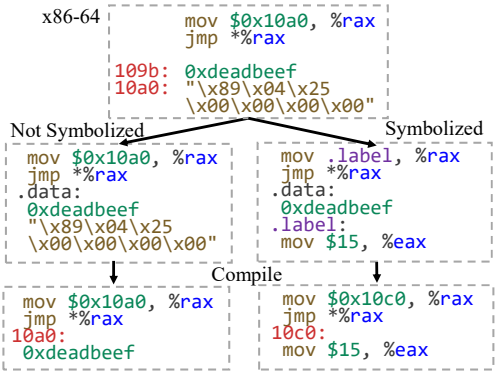


Figure 1. An example of symbolization.

Symbolic Execution. Symbolic execution is a powerful program analysis technique that verifies program properties for all possible inputs [13, 20, 85]. Instead of using concrete data, it executes a program with symbolic values and tracks them throughout the program. When a branch is encountered, it explores both paths separately, generating a set of symbolic constraints that represent the conditions required to reach specific program states. These constraints can be evaluated by SMT solvers [32, 71] to check for satisfiability and verify program properties. Symbolic execution has proven effective for analyzing binary code [86, 88, 102], offering a rigorous approach to verify translated binary code.

2.2 Large Language Models

Recent breakthroughs in LLMs have demonstrated their effectiveness in various natural language processing tasks like machine translation and text summarization [21, 108, 118]. Their application has also extended to code-related tasks, such as code completion and compilation [26, 48, 121]. The power of LLMs stems from their extensive training on high-quality data, which cultivates sophisticated reasoning and inference capabilities for tasks that traditionally required significant human expertise. Inspired by this, our work explores the potential of LLMs for cross-architecture instruction translations, yielding promising results. However, LLMs are known for their unpredictability; they can produce plausible but incorrect outputs with no guarantee of correctness [57, 117]. To mitigate this risk and prevent the generation of erroneous binaries, we introduce a crucial rigorous semantic equivalence check step, detailed in Sec. 4.4.

2.3 Binary Translation

Binary translation has been a long-standing topic of academic research [97, 106]. This paper focuses specifically on user-level binary translation, which handles general applications [15, 29, 61, 98, 107], as opposed to system-level binary translation that targets operating systems (OSes) [61].

General Workflow. A typical binary translator begins by disassembling the binary software to extract its assembly code. Then, depending on the translator's design, it can directly translate the assembly code to the host ISA, or lift assembly to a platform-independent IR, which is then lowered to host assembly using established compiler infrastructures (e.g., LLVM). For efficiency, instructions are often organized and translated in translation blocks (TBs) [17, 69] that are typically stored in a cache to avoid redundant translations.

Terminology. In this paper, the term *binary translators* refers to tools for translating and migrating binary software across architectures. However, existing tools use distinct technical approaches with different IRs and translation styles employed, significantly impacting their capabilities and performance. To differentiate these approaches clearly, we extend the terminology developed in recent studies [74, 83] and categorize them into four types:

Concise IR Lifter. These tools aim to recover LLVM IR that closely resembles the IR code generated by compiling source code. They *lift* low-level binary code to high-level, concise, and human-readable IR, attempting to restore source-level information (e.g., variables, types, and symbols). Notably, works like Mctoll [114], RetDec [12], and Rev.ng [35, 36] have dedicated significant effort in this direction. One subsequent advancement, Lasagne [91], has further enhanced Mctoll's capabilities for concurrent programs. Ideally, such lifters offer high flexibility, as the lifted IR can naturally provide cross-architecture portability and can be readily analyzed and optimized using established LLVM-based tools.

However, perfect IR lifting faces multiple fundamental challenges, including incomplete control flow graph (CFG) recovery [36, 79, 95] and the loss of high-level abstractions like functions and variables in binary code [51, 113]. Consequently, these tools often do not prioritize functionality preservation; instead, they are primarily used for automated program analysis, binary similarity comparison [78], and reverse engineering tasks like decompilation [77]. Our evaluation confirms that they are not suitable for reliable cross-architecture binary translation.

Emulational IR Lifter. In light of the difficulties in recovering high-level abstractions from binary code, another line of research focuses on emulating the binary's semantics at the IR level. These lifters translate each machine instruction into a sequence of IR statements that precisely mimic its effects on a virtual CPU in lifted IR code [74, 83]. For example, McSema [37, 81] maintains a dedicated structure in lifted IR to represent the CPU state, including registers and flags. A similar paradigm is employed by dynamic lifters like Instrew [40, 41] and BinRec [3]. Two subsequent variants of BinRec, named WYTIWYG [83] and Polynima [34], while enhancing its support for stack-layout recovery and concurrent programs, still fall into this emulational category.

Table 1. Relevant binary translators investigated.

	Type	IR Used	Functional Fidelity	Flexibility*	Primary Goal
Mctoll [114]	Static	LLVM	Low	High	Readable Concise IR
RetDec [12]	Static	LLVM	Low	High	
Rev.ng [36]	Static	LLVM	Low	High	
McSema [81]	Static	LLVM	Medium	Medium	High-Fidelity IR
Instrew [41]	Dynamic	LLVM	High	Medium	
BinRec [3]	Dynamic	LLVM	High	Medium	
QEMU [15]	Dynamic	TCG	High	Low	Reliable & Portable
ExaGear [58]	Dynamic	None	High	Low	High Performance
LLMSbt	**Static**	None	High	Low	

* In this context, flexibility is defined as the ease of porting to new architectures and applying standard compiler analyses and optimizations.

Despite guaranteeing a high degree of functionality, this approach sacrifices the benefits of LLVM IR as a strongly-typed IR. The lack of high-level semantic information, such as types, hinders automated analysis and optimization [74, 83]. Furthermore, this emulational strategy often introduces extra code in translated binary, resulting in inefficiency [74, 83]. Given that static lifting struggles with complete CFG recovery (McSema relies on the de facto disassembler IDA Pro [53] for CFG recovery and disassembly), dynamic lifters like Instrew defer IR compilation and optimization to runtime, which further exacerbates performance issues.

Low-level IR Translator. This category of works [15, 90], which includes CPU emulators like QEMU, forgoes the cross-architecture benefits of LLVM IR in favor of a lower-level IR that more closely mirrors machine code. For decades, QEMU [15] has been widely adopted in industry and academia due to its high reliability.

Recent studies, however, indicate that QEMU can be over 10× slower than native execution [112]. This performance penalty stems from a fundamental trade-off between scalability and portability. To support a variety of architectures, QEMU uses a lower-level IR (TCG IR) [15, 17]. Each CPU instruction is decomposed into several micro-operations in TCG IR, which are implemented in high-level programming languages and compiled into native host machine code. While this shields developers from the intricacies of complex ISAs and hardware specifics, it limits low-level optimization opportunities and introduces inefficiencies compared to hand-tuned assembly, leading to substantial overhead.

Assembly Translator. To achieve maximum performance, industrial dynamic translators like ExaGear [58] bypass IRs and translate guest instructions directly into native host instructions, leveraging hardware features and delivering near-native performance. The downside is that developing the translation logic is labor-intensive, error-prone, and typically tailored for specific platforms, limiting portability and reusability. Moreover, the dynamic paradigm they employed also introduces substantial runtime overhead.

LLMSbt mitigates these issues by automating the translation process using LLMs. Its static paradigm allows us to incorporate LLMs and rigorous verification into the translation process. Moreover, by decoupling the translation framework from translation rules, we transform translation into a data-driven process. As a result, developing the translator requires minimal expert knowledge, as rules are mostly extracted from LLM-generated examples.

It is important to note, however, LLMSbt is not intended to replace the mature binary translators developed by industry over the years. Binary translation remains a highly complex field requiring substantial engineering effort (e.g., QEMU consists of over one million LoC, and Huawei has spent more than five years refining ExaGear). Because there are still many unresolved problems requiring further research, this work primarily contributes new concepts and a novel translation paradigm to the community.

Previous Works to Compare. To position our work, Table 1 presents a qualitative synthesis and taxonomy of representative translators. Specifically, the classification is grounded from a recent research [74], whose observations have been confirmed by subsequent studies [83, 119]. We adopt this established terminology and taxonomy while expanding our scope to include four additional tools (Rev.ng, Instrew, QEMU, and ExaGear) for a more thorough discussion. Within this taxonomy, functional fidelity refers to the strictness of semantic preservation. High fidelity ensures full recompilability, while low fidelity prioritizes static analysis and code comprehension over executability. Although McSema faithfully lifts all individual instructions for small-scale benchmarks [30], its performance in our evaluation justifies a medium fidelity rating (as detailed in Table 5). Flexibility follows a similar categorization. High flexibility indicates the recovery of strongly-typed IR compatible with standard compiler passes, while low flexibility refers to methods tied to hardware specifics such as direct assembly translation. Emulational IR lifters occupy a middle ground, producing IR that allows for cross-target lowering but lacks support for standard compiler analysis.

Table 1 highlights the functional and structural limitations of current approaches. Our investigation found that while BinRec shows promising performance, it is designed for same-architecture recompilation and optimization and is unsuitable for cross-architecture translation because it must concatenate the lifted binary with the original. Concise IR lifters, despite backing from notable technology companies, often fail to deliver fully functional results (more details of their failures in the Appendix A.1). In fact, tools like RetDec and Rev.ng primarily aim at software decompilation, treating translation failures and semantic mismatches as low-priority issues [100]. Consequently, our evaluation mainly compares LLMSbt with state-of-the-art in each category: McSema, Instrew, QEMU, and ExaGear. Sec. 3 further discusses more

design choices and rationales of LLMSbt, aiming for high reliability and runtime efficiency.

We also acknowledge recent advancements in this domain that solve specific complexities, such as concurrent programs [34, 46, 91] and stack-layout recovery [83], in modern software. They are not separately listed as they are primarily built upon works already covered in Table 1, with their core designs remaining unchanged. We further discuss them in Sec. 8.

3 Motivation

As discussed in Sec. 2.1, static disassembly struggles to resolve indirect control flow targets and symbols missed during compilation. While some static translators employ mature commercial disassemblers like IDA Pro [53] to partially mitigate these issues [81], they remain brittle in general scenarios. This unreliability discourages the adoption of static disassembly in translators where execution correctness is paramount. Consequently, industrial binary translators typically rely on dynamic disassembly, which necessitates a dynamic translation paradigm and introduces significant runtime overhead, preventing them from producing portable binaries, i.e., a guest binary can only execute on platforms with the translator installed.

To overcome these fundamental limitations, we introduce a new translation paradigm centered around the following major conceptual contributions:

1) Static Assembly-to-Assembly Translation. As noted in Sec. 2.1, recent advancements in binary rewriting present a *unique opportunity* for static binary translation. Leveraging these binary rewriting techniques, we propose a novel static translation paradigm that translates assembly code, bypassing the challenges of IR lifting and ensuring high reliability. Furthermore, the static paradigm reduces runtime overhead and produces portable, native-like binaries, opening new avenues for research and downstream optimization.

2) Automated Translator Development. State-of-the-art industrial binary translators like Rosetta 2 [8] and ExaGear [58] are closed-source and tailored to specific hardware and OSes, lacking support for general AArch64 processors. Meanwhile, existing open-source translators [3, 41, 81] are unsuitable for our static assembly-to-assembly paradigm because they lift assembly to IRs (as introduced in Sec. 2), which inevitably introduces unnecessary overhead. Building a new translator from scratch is a formidable task that requires a tremendous engineering effort. The absence of open, efficient translators for general AArch64 platforms hinders further research.

To establish a practical foundation for future research, we introduce a novel binary translation pipeline. Specifically, we propose using LLMs to automatically generate translation results, replacing the need for human experts. This approach is economically viable due to the affordability of modern LLMs. Furthermore, since many instruction variants share similar translation patterns, previously generated translations can be parameterized and reused to accelerate future translation.

3) Scalable Semantic Verification. CPU instruction semantics are often intricate, and even experienced developers may mistranslate subtle semantic details. Modern ISAs, particularly complex instruction set computing (CISC) architectures like x86-64, contain a vast number of instructions, forcing developers to spend significant time consulting manuals and designing test cases to ensure accurate translation. This is a tedious and error-prone process that still does not guarantee correctness. While methods like symbolic execution can rigorously verify binary code, they often face scalability issues due to path explosion [92].

Verifying translation correctness is a difficult but essential challenge. To address this, we introduce an efficient static verification method. Instead of verifying an entire complex program, we split assembly code into small, manageable snippets that can be easily verified with symbolic execution and then reassembled. This method provides instant feedback to human experts or LLMs, streamlining the development of robust binary translators.

4 Design

This work formulates a new translation paradigm, outlining a concrete methodology for developing a high-performance, portable binary translator empowered by LLMs. We present LLMSbt, a static binary translator for migrating x86-64 software to AArch64 platforms. This section presents a design overview and then examines each component in detail.

4.1 Overview

As shown in Fig. 2, the translation process starts with standard disassembly. What distinguished our approach is the use of advanced binary rewriting techniques to transform disassembled code into a symbolic, reassembleable form. We then split the binary code into snippets (①), with each snippet representing a code block in a Static Single Assignment (SSA)-like format [27, 28, 101]. For clarity, we use "SSA" throughtout the paper to refer to our adapted concept, which differs slightly from the traditional SSA used in compiler research (as explained in Sec. 4.2). The SSA representation enables independent translation of each snippet. For each snippet, we first query the translation rule database for existing, reusable translations (②). If no cached translation can be reused, our LLM-powered engine translates the snippet (③); otherwise, the existing translation rule in the database is applied. Once snippets are translated, we employ symbolic verification to ensure the semantic correctness of the translated snippets (④). Finally, we stitch translated snippets into complete, compilable assembly code (⑤ and ⑦). Moreover, to minimize the cost of repeatedly querying the LLM for similar code, we further parameterize translation results

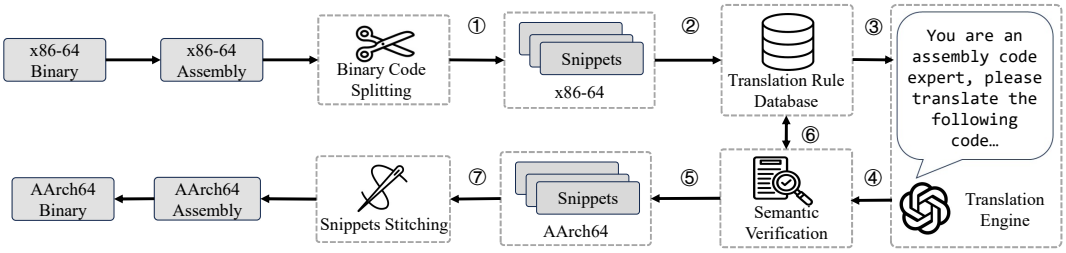


Figure 2. Overview of LLMSbt.

into reusable rules for future translations (⑥). The following subsections elaborate on each of these components.

4.2 Binary Code Splitting

Translating entire programs or functions poses significant hurdles to LLMs and verification. The extensive context length required, coupled with intricate data and control flows in assembly code, often exceeds the reasoning capabilities of LLMs. Additionally, verifying the correctness of large-scale translated code is a difficult task. Drawing inspiration from the divide-and-conquer strategy [96], we propose a fine-grained translation approach. Our method focuses on translating small code snippets formatted in SSA form. The snippets are generally concise enough for LLMs to comprehend and process. The SSA form simplifies data and control flows, minimizing potential errors introduced by LLMs and enabling the semantic verification of the translation.

Adapted SSA. Through static analysis, we break down each basic block into SSA snippets. Here, SSA requires that within each snippet, a variable is assigned a value at most once (i.e., single assignment). Unlike the traditional SSA concept used in compiler research, our adaptation includes cases where a variable might be used before definition. This distinction arises because, unlike high-level languages, variables in assembly (such as registers and memory) are inherently defined by the hardware state and are never truly undefined.

Splitting. This step starts with symbolized assembly code. The splitting process is guided by the core principle of single assignment to reduce complexity. As illustrated in Alg. 1, we linearly scan the assembly instructions (lines 6, 13), tracking which variables are assigned at each instruction. Whenever a previously assigned variable is rewritten (line 7), we split preceding instructions into a snippet and reset the track (lines 8–10). This process continues until the end of a basic block, where control flow transfers naturally conclude a snippet. Due to the intangibility of statically determining the address of memory accesses, we treat all memory as a single variable for simplicity. This means each snippet can have at most one memory access. This step yields a list of SSA snippets ready for independent translation.

Algorithm 1: Splitting Basic Block to SSA Snippets

```
1  Function Split(BB_instrs: a list of instructions from a basic block)
2      S ← ∅;
3      CurrentSnippet ← [];
4      AssignedVars ← ∅ ;          // variables assigned in
         CurrentSnippet
5      foreach instr in BB_instrs do
6          DefinedVars ← GetDefinedVariables(instr);
7          if AssignedVars ∩ DefinedVars ≠ ∅ then
8              Add CurrentSnippet to S;
9              CurrentSnippet ← [instr];
10             AssignedVars ← DefinedVars;
11         else
12             Append instr to CurrentSnippet;
13             AssignedVars ← AssignedVars ∪ DefinedVars;
14     if CurrentSnippet is not empty then
15         Add CurrentSnippet to S;
16     return S ;          // Output: A list of SSA snippets
17 Function GetDefinedVariables(instr)
18     D ← ∅;
19     if instr writes to memory then
           // For simplicity, all memory is treated as
              one variable.
20         D ← D ∪ {MEMORY};
21     R_dest ← destination registers of instr;
22     D ← D ∪ R_dest;
23     return D;
```

4.3 LLM-Empowered Translation

Each x86-64 snippet is then translated to AArch64 individually with an LLM. However, directly translating assembly code with LLMs can be error-prone, particularly with complex CISC ISAs like x86-64. While LLMs generally possess knowledge of relevant instructions, they can produce incorrect translations due to insufficient context or limited reasoning capabilities. To meet the strict correctness requirements of binary translation, we design a domain-specific prompt architecture that adapts recent advances in prompt engineering to the constraints of assembly language.

Semantic-Grounded Reasoning. First, LLMs can exhibit shallow reasoning, opting for superficial interpretations rather than performing a deep semantic analysis of the code. To

address this issue, we implement a specialized task-splitting strategy, effectively adapting chain-of-thought (CoT) prompting [109] to the binary domain, which instructs the LLM to interpret the assembly code before translating it. This preliminary interpretation step grounds the LLM's understanding of code's semantics, promoting more accurate and deliberate reasoning. We strategically choose not to verify this intermediate interpretation, acknowledging that hallucinations can occur at any stage. Instead, we treat interpretation as an auxiliary cognitive scaffold to prime the model, focusing our rigorous formal verification efforts exclusively on the final executable output (see Sec. 4.4). For example, as shown in Fig. 3, the LLM initially misinterprets an instruction as moving the address of label `.LCd0` into `x1`. However, by first prompting the LLM for a semantic interpretation, the LLM can produce the correct translation, which loads a 64-bit integer from the memory indicated by `.LCd0`.

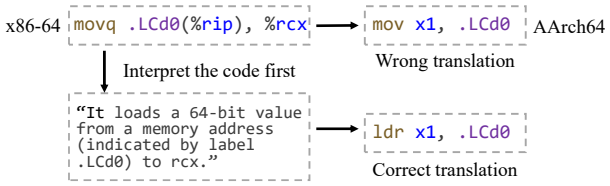


Figure 3. An example of semantic-grounded reasoning.

RAG-Enhanced Translation. Second, LLMs may generate syntactically invalid code despite correct semantics understanding, often due to insufficient context. We repurpose retrieval-augmented generation (RAG) [25, 45] to solve the brittleness issue of assembly code synthesis. We retrieve similar, correct translation examples from our database based on text similarity and inject them as architectural context. These examples provide the LLM with precise, compilable syntax patterns, helping LLM to produce valid code. Fig. 4 shows an example where the initial translation (top right) is non-compilable because an immediate value (−544) exceeds the allowable range. By providing a textually similar, valid example, the LLM learns the correct pattern and generates a valid translation (bottom right).

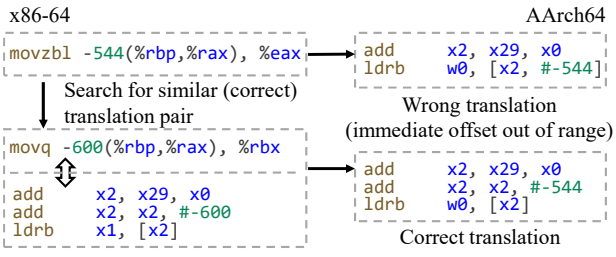


Figure 4. An example of RAG-enhanced translation.

Iterative Refinement. Finally, to fix any remaining errors, we introduce an iterative process to refine and correct the results. A key contribution of our approach is replacing standard heuristic-based refinement with a feedback loop grounded in formal verification. Specifically, we perform a rigorous semantic verification using the `angr` symbolic execution engine [102] and the Z3 solver [32] (details in Sec. 4.4). If any errors or semantic inconsistencies are found, the precise feedback (i.e., formatted output from the symbolic execution engine) is passed back to the LLM. The LLM is then prompted to refine the translation based on this feedback, effectively combining the reasoning ability of LLMs with the rigor of formal proofs. The newly refined translation is then verified with symbolic execution again. This iterative cycle continues until the translation passes semantic verification or a predefined iteration limit (set to 5) is reached.

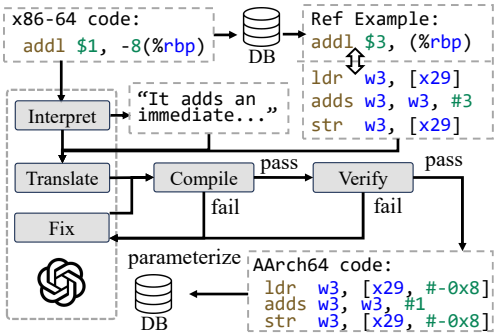


Figure 5. Workflow of iterative refinement.

As a result, we establish an iterative pipeline for our translation engine, as shown in Fig. 5. While the interpret, translate, and fix steps are LLM-powered, we use the GCC compiler and `angr` for compilation and verification. Our evaluation confirms that this specialized integration of prompt engineering and formal verification forms a necessary innovation to enable reliable, LLM-based binary translation.

4.4 Semantic Verification

The semantic verification process ensures that the translated code is semantically equivalent to the original. Two snippets are considered semantically equivalent if they always produce identical outputs for the same inputs.

Variables. This equivalence requires a one-to-one mapping between input and output variables, where corresponding variables hold the same value after execution. Variables in assembly code include registers and memory, while memory is treated as a special variable that encompasses both its value and the address being accessed. Furthermore, instruction execution often updates the flags register, which subsequent instructions (especially branches) may utilize. Since `angr` treats flags as specialized registers, we can verify them using the same methodology for general-purpose registers. While x86-64 defines six status flags, two of them (AF and PF) are primarily retained for backward compatibility [47, 94]. Our analysis confirmed that these two flags are

unused in our evaluation set. The remaining four flags have direct counterparts in AArch64 and can be mapped reliably.

Symbolic Execution. To verify semantic equivalence, we first perform symbolic execution on both the x86-64 and translated snippets. We assign initial symbols to all input variables and conduct symbolic execution to derive symbolic expressions for each output. Crucially, the snippets are ensured to have only one valid execution path without branches. This design choice deliberately bypasses the path explosion problem that complicates symbolic execution in general scenarios. Fig. 6 demonstrates an example of symbolic execution results, where initial symbols are marked with an `_init` suffix. The verification requires finding a valid mapping between registers that satisfies the aforementioned equivalence property. Using brute-force search, we explore possible register mappings and employ the Z3 constraint solver [32] to check if a given mapping satisfies the equivalence property. The small number of variables makes this search computationally efficient. If a valid mapping is found, the two snippets are deemed semantically equivalent.

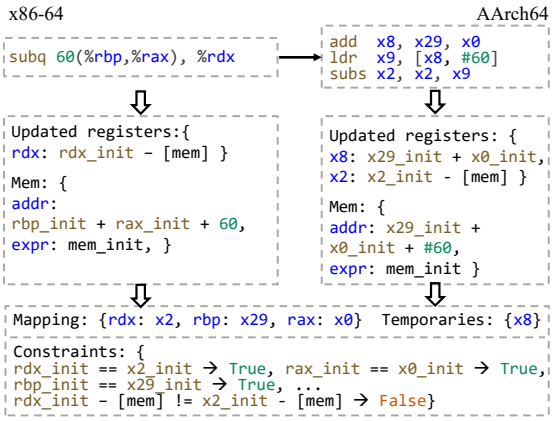


Figure 6. An example of symbolic execution results.

Equivalence Checking. Assuming the x86-64 snippet contains variables $V_A = \{v_{A_1}, v_{A_2}, v_{A_3}, \ldots, v_{A_{nA}}\}$, and the AArch64 snippet uses variables $V_B = \{v_{B_1}, v_{B_2}, v_{B_3}, \ldots, v_{B_{nB}}\}$, let $E(v_i)$ denote the symbolic expression of variable v_i, with $E_{init}(v_i)$ as its initial symbolic expression. Given a mapping $M : V_A \rightarrow V_B$, for each variable $v_i \in V_A$, we first establish the initial conditions by adding the constraint $E_{init}(v_i) = E_{init}(M(v_i))$ to the solver. This ensures that corresponding variables start with the same initial symbolic expression. After that, for each variable $v_i \in V_A$, we check if $E(v_i) \neq E(M(v_i))$ is unsatisfiable with the solver. If it is, we add $E(v_i) = E(M(v_i))$ to the solver and proceed to the next variable. This ensures that symbolic expressions of corresponding variables *must* be equal after execution. Once all variables are validated, the mapping is considered to satisfy the equivalence property. This process naturally handles temporary registers that may be introduced in the AArch64 snippet, as x86-64 instructions are often decomposed into multiple simpler RISC operations.

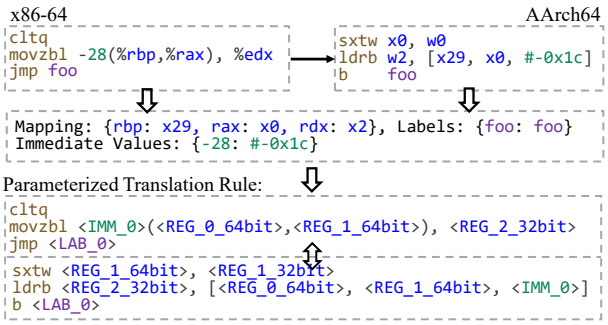


Figure 7. An example of a parameterized translation rule.

4.5 Translation Rule Parameterization

Querying an LLM is time-consuming and inefficient, especially for repeated similar code. To address this, we parameterize verified translation results into reusable rules.

Parameterization. Specifically, we observed that snippets with similar structure, albeit with different operands, often yield translation results with the same sequence of AArch64 instructions. Therefore, we abstract concrete operands (e.g., registers and immediate values) into parameters, extracting generalized snippet skeletons. This allows the reuse of a verified translation rule for multiple similar x86-64 snippets without repeated LLM queries. Furthermore, the parameterized translation rules are context-independent and can guarantee consistent functionality when reassembled.

Fig. 7 illustrates an example of a parameterized translation rule. The translation at the top is semantically correct and verified. We replace its concrete operands with unique parameter identifiers (composed of its type and ID) based on the mapping derived from verification. These parameterized rules, along with original snippets, symbolic verification results, and mappings, are stored in the rule database.

Database Querying. Before translating a new x86-64 snippet, we first parameterize it and query the database for matched rules. A match requires that the snippet's instruction, parameter types, and sequence are identical to a rule in the database. Upon finding a match, the rule is applied by replacing parameters with concrete operands. Otherwise, the snippet is sent to the LLM. For consistency, we conduct the same parameterization process on translation results produced by LLMs. In practice, as the database grows, these rules increasingly supplant the role of LLMs, shifting translation into an efficient data-driven process. Consequently, ③ and ④ in Fig. 2 become less frequent, with most snippet translations being handled by reusing rules in the database (⑥), reducing the time and cost of LLM queries.

Concretization. Once all snippets are translated, the resulting rules, whether from the database or generated by parameterizing LLM outputs, need to be concretized before they can be stitched into complete assembly code. This process reinserts specific operands back into the translated snippets. While immediate values and labels can be directly extracted from x86-64 snippets, register concretization demands a consistent mapping across snippets (i.e., a specific x86-64 register must always map to the same AArch64 register). For simplicity, we employ a fixed mapping to define the correspondence between x86-64 and AArch64 registers, as detailed in Table 2. This is feasible without causing register conflicts because AArch64 offers more general-purpose registers than x86-64. The remaining AArch64 registers are utilized for temporary values introduced during translation.

Table 2. The mapping used to concretize translation rules.

x86-64	AArch64	description
rdi	x0	Function arguments
rsi	x1	
rdx	x2	
rcx	x3	
r8	x4	
r9	x5	
rax	x8	Return value
rbx	x9	Callee saved
r10-r15	x20-x25	temporary registers
rbp	x29	Frame pointer
rsp	sp	Stack pointer

4.6 Snippets Stitching

The final step is to stitch concretized snippets into an executable assembly program. This offline process introduces no runtime overhead and primarily involves reconstructing the program's control flow and managing Application Binary Interface (ABI) differences with function calls.

Control Flow Reconstruction. We arrange the translated snippets in the original sequence as their x86-64 counterparts. We iterate through each basic block and emit AArch64 assembly code for each snippet. Jumps and branches are resolved by mapping all target addresses to appropriate labels. For instance, an `jne .label` is translated into a corresponding `b.ne .label` in AArch64, where *.label* points to the equivalent location in the new code.

Stack Alignment Management. Moreover, AArch64 enforces stricter stack alignment requirements than x86-64. A common issue arises when the prologue code of an x86-64 function pushes an 8-byte value, which can misalign the 16-byte boundary required by AArch64. To resolve this, we employ a standard AArch64 assembly programming technique [18]. Upon entering a function, we substitute the stack pointer (`sp`) with a free register (e.g., `x28`). As shown in Fig. 8, using `x28` in the store (`str`) instruction allows the original stack layout to be preserved while ensuring that `sp` remains 16-byte aligned. The stack pointer is restored upon exiting the function, maintaining proper stack discipline.

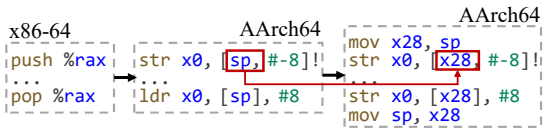
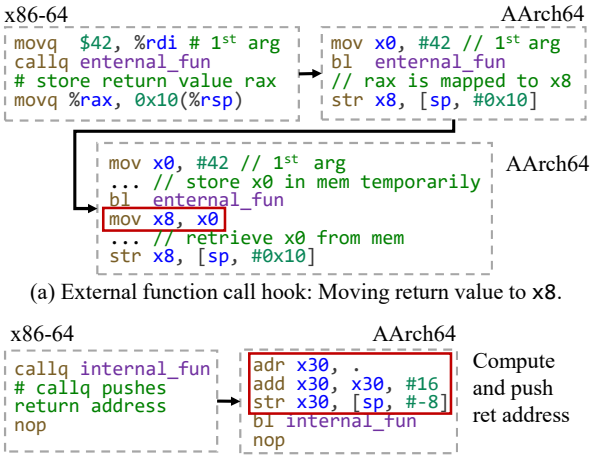


Figure 8. Stack Alignment Management.



(a) External function call hook: Moving return value to `x8`.

(b) Internal function call hook: Pushing return address explicitly.

Figure 9. Examples of extra code inserted.

Correctness. The correctness of the stitched program hinges on the principle of *compositional verification* [24]. Each x86-64 snippet is translated into a semantically equivalent AArch64 snippet. The use of SSA form makes data flow explicit through def-use chains, allowing each snippet to be treated as a verified, context-independent function. Since each translated snippet produces identical output as the original snippet, the correctness of stitched programs can be reasoned. However, the correctness guarantee relies on two main assumptions:

1) No Implicit Cross-Snippet Dependencies. We assume that there are no hidden side effects or dependencies between snippets that are not captured by the explicit control and data flow. This holds for standard application code but excludes highly specialized cases like self-modifying code.

2) Mediated ABI Compatibility. We assume that external libraries are effectively managed, and ABI incompatibilities are resolved with additional expertise. That requires all necessary libraries to be available on the AArch64 host, and our hooking mechanisms (introduced below) are applied to function calls to mediate the incompatibilities.

Function Call Hooks. While stitching, we hook calls to resolve ABI differences between the x86-64 and AArch64, which arise from their distinct architectural designs. While mitigating such issues requires domain expertise and extra

code, this is a standard practice for cross-architecture binary translation and commonly seen in related binary analysis works [38, 63, 79, 114]. Specifically, our hooks address inconsistencies in calling conventions for external calls and return address handling for internal calls.

Calling Convention. Since translated code follows the x86-64 calling convention, it requires no special handling for arguments within translated functions. However, differences in the ABI must be addressed at the boundaries between translated and native code (e.g., external library calls). First, the return value and first argument in x86-64 are in separate registers (`rax` and `rdi`), whereas in AArch64, both are handled by the `x0` register. Our implementation maps `rax` and `rdi` to different registers (`x8` and `x0`, respectively). As demonstrated in Fig. 9(a), this is addressed by preserving and retrieving the first argument before and after a function call, and then transferring the return value from `x0` to `x8` after the call. Second, x86-64 passes the first six arguments via registers (`rdi-r9`), with subsequent arguments on the stack. AArch64, however, uses registers `x0-x7` for the first eight arguments. Our fixed register mapping already aligns x86-64's `rdi-r9` with AArch64's `x0-x5`. To ensure correct argument passing for functions with more than six arguments, we move the seventh and eighth arguments from the stack into registers `x6` and `x7` before the call.

Return Address. Additionally, x86-64 and AArch64 also manage return addresses differently. In x86-64, the `call` instruction automatically pushes the return address onto the stack, while in AArch64, the `bl` instruction stores the return address in the link register (`x30`). Thus, an internal function call from translated code will leave the return address in `x30`, which can be overwritten by subsequent calls. Our solution, shown in Fig. 9(b), hooks the call to explicitly push the return address to preserve it and pop it back into `x30` before returning, ensuring the integrity of the call stack.

Threats to Validity. The issues resolved in the stitching process are unique for assembly-level translators, as lifters typically compile lifted IR into binary without diving deep into architectural differences. While domain expertise is required, the core problems and solutions are common and well-documented [10, 11]. Nevertheless, the validity of stitching faces potential threats:

Complex Data Types. While we address common ABI discrepancies, issues may arise with specialized data types (e.g., floating-point values and vectors) or aggregate types, whose passing can differ largely between architectures.

Non-Standard ABIs. Customized libraries may employ non-standard or highly specialized calling conventions not covered by our method, leading to incorrect argument passing or return address handling. In such cases, the libraries should be translated as well.

While solving these potential issues may require additional expertise and effort, the root causes can generally be identified by differentially debugging translated and original binaries. Solutions can then be derived from established assembly programming practices and ABI documentation.

5 Implementation

The core components of our research prototype, LLMSbt, consist of roughly 6K LoC in Python (excluding comments and blank lines), leveraging the `angr` symbolic execution engine [102] and GPT-4 [82] as the underlying LLM. For disassembling and symbolizing binary code, we chose `RetroWrite`, an open-source binary rewriter that has been extensively evaluated in prior studies [38, 64]. All binaries were compiled with GCC 11.4.0 on an x86-64 machine running Ubuntu 22.04, equipped with an Intel Core i7-9700K CPU and 32 GB RAM. The translated binaries were executed on a development board featuring an ARMv8.2-based rk3588 processor [1]. For comparison with ExaGear, which is compatible only with Huawei-manufactured CPUs, we ran translated binaries on a Huawei Cloud virtual machine powered by a Kunpeng 920 processor (ARMv8.2) [54].

Target Software. Our current implementation targets user-level single-threaded C programs compiled without anti-analysis techniques like obfuscation or self-modification. The evaluation mainly covers a subset of x86-64 ISA, including core general-purpose instructions and frequently used SSE instructions. Support for floating-point operations is limited, as we currently exclude x87 FPU instructions. Limitations and future works are further discussed in Sec. 7.

6 Evaluation

This section evaluates the performance of LLMSbt by answering the following research questions:

- **RQ1:** How many translation rules were collected, and what were the development costs of LLMSbt?
- **RQ2:** How does the runtime performance of binaries translated by LLMSbt compare to existing open-source translators and the industrial dynamic binary translator ExaGear?
- **RQ3:** What factors affect the execution efficiency of translated binaries?

6.1 Characteristics of Translation Rules

To answer **RQ1**, this section details the characteristics of the 3, 149 rules collected by LLMSbt. These rules encompass 3, 756 x86-64 instructions and 7, 784 AArch64 instructions in total. On average, each rule maps 1.2 x86-64 instructions to 2.5 AArch64 instructions, reflecting our observation that converting CISC instructions to RISC instructions often requires multiple instructions. Meanwhile, as shown in Table 3, about 95% rules have three or fewer x86-64 instructions, while the number of AArch64 instructions mostly ranges from one to five. Generally, shorter snippets are more likely to match similar snippets and easier for LLMs to translate. While longer snippets likely expose richer optimization opportunities, no

significant correlation was observed between snippet length and LLM failures. However, it is worth noting that the LLM did not achieve perfect coverage. Consequently, we manually provided an additional 68 rules for cases where the LLM failed, the details of which are discussed in Sec. 7.

Table 3. The number of translation rules of different lengths.

#Inst	1	2	3	4	5	6	≥ 7	Sum
x86-64	2,177	524	297	144	4	2	1	3,149
AArch64	689	677	678	585	279	112	129	3,149

The rule collection process was highly cost-effective. The entire process, including iterative corrections, consumed approximately $2,036K$ tokens, totaling less than $130. With an average requirement of under 30 seconds and roughly $0.04 per rule, these results highlight the economic viability of our approach. In terms of instruction set coverage, the collected rules cover 148 unique x86-64 instructions and 102 AArch64 instructions. While modern CPUs support a vast number of instructions, many are highly specialized for specific hardware features (e.g., AVX512 and SGX) or are rarely used. Translating these hardware-dependent instructions presents unique challenges that are outside the scope of this research. This work primarily pioneers a new paradigm for binary translation by focusing on 259 core general-purpose instructions and frequently used SSE instructions [2, 31, 76]. For comparison, QEMU contains 283 code generation functions for x86-64 [87], each of which translates one x86-64 instruction into IR. The ability of LLMSbt to collect a significant set of rules at a very low cost, covering a substantial portion of the core instruction set, shows high potential and research value of this approach. Interestingly, during our evaluation, we found a program where QEMU failed (see Table 5). QEMU throws an "Illegal Instruction" error when translating the `shlx %rsi, %rdx, %rdx` instruction. This suggests that even widely-used translators like QEMU may still contain flaws, underscoring the motivation of our work.

6.2 Performance Overhead

Test Cases. We use a comprehensive suite of benchmarks, including CHStone [49], Coremark-Pro [39, 44], and SPEC CPU benchmark [19, 52]. We also include two widely used real-world projects: `Zydis` [122], a fast x86/x64 disassembler library, and `yyjson` [59], a high-performance JSON library. Our evaluation encompassed a total of 23 diverse programs, detailed in Table 4 and Table 9. This test suite ensures a thorough assessment of LLMSbt's effectiveness across a range of complex instructions. We empirically consider a translation correct when the translated binary exhibits consistent input-output behavior on the test cases.

Table 5 compares the performance overhead of LLMSbt with QEMU, Instrew, and McSema on the rk3588 platform. Static lifters like Mctoll and RetDec were excluded because they failed to process any test case, a limitation consistent with findings in prior research [34, 74, 83]. The results reveal that LLMSbt consistently and significantly outperforms other translators. Notably, McSema failed on 11 of 23 cases, while both Instrew and QEMU failed on the `xz` case. These findings underscore substantial performance and robustness advantages of our approach over traditional methods.

Table 4. Statistics of evaluated software.

	Source LoC	#Func		Source LoC	#Func
adpcm	887	15	loops	9.5K	328
gsm	634	12	parser	18K	310
mips	388	1	radix2	7K	289
motion	1.1K	14	zlib	21.5K	396
aes	1.1K	11	bzip2	8.1K	70
sha1	1.4K	8	xz	35.6K	390
blowfish	1.8K	6	mcf	4K	43
jpeg	3.3K	29	libquantum	4.3K	98
cjpeg	19.5K	416	hmmer	38.3K	494
sha256	7.2K	283	Zydisn	18K	161
linpack	7.5K	290	yyjson	12K	92
nnet	8.2K	288	Sum	219.4K	4,044

Moreover, QEMU and Instrew occasionally exhibited substantial overhead (over 20×). Our investigation revealed that these programs' test suites consist of low-workload cases, causing frequent program restarts. Since translated code is not retained between runs, each restart forces a repeated translation, contributing to the high overhead. Moreover, Instrew's reliance on LLVM for runtime optimization and code generation further adds to this overhead. Nevertheless, it is important to note that Instrew's novel design is a significant contribution, offering an efficient and reliable system that substantially outperforms QEMU in high-workload scenarios. In contrast, static approaches like McSema and LLMSbt turn translation into a one-time offline effort, producing portable binaries that execute natively. However, McSema's low success rate highlights the inherent difficulties of static IR lifting. We attribute LLMSbt's performance and robustness advantages to its static assembly-to-assembly translation paradigm, which bypasses the daunting binary analysis challenges while minimizing runtime overhead.

Table 6 further compares LLMSbt's performance against ExaGear. While both tools exhibit strengths, LLMSbt demonstrates consistently stable and superior performance overall. Similarly, ExaGear's performance suffers from significant overhead in low-workload scenarios. Given that ExaGear is a mature, heavily optimized commercial product developed by a leading chip manufacturer, these results highlight the remarkable potential and research value of our approach.

6.3 Performance Analysis

To better understand why LLMSbt's performance varies across different cases, we conducted detailed profiling on three typical algorithms from a popular C library [56]: `avl-tree`,

Table 5. Performance overhead comparison on rk3588 versus native AArch64 execution. Cases with no lifted code, compilation failures, or inconsistent results are marked "failed".

Testcase	QEMU	Instrew	McSema	LLMSbt
adpcm	33.57×	>100×	1.94×	**1.22×**
gsm	33.94×	>100×	1.58×	**1.23×**
mips	34.08×	>100×	1.90×	**1.26×**
motion	33.67×	>100×	1.87×	**1.24×**
aes	33.74×	>100×	1.84×	**1.22×**
sha1	29.84×	>100×	2.29×	**1.24×**
blowfish	29.16×	>100×	2.16×	**1.26×**
jpeg	23.03×	>100×	3.62×	**1.24×**
cjpeg	3.52×	>100×	failed	**1.36×**
sha256	32.69×	>100×	1.72×	**1.19×**
linpack	8.94×	90.1×	failed	**1.26×**
nnet	15.24×	33.3×	4.67×	**1.86×**
loops	6.51×	76.4×	failed	**1.27×**
parser	2.97×	83.7×	1.88×	**1.03×**
radix2	25.58×	>100×	46.04×	**1.27×**
zlib	17.61×	>100×	failed	**1.15×**
bzip2	12.82×	1.32×	failed	**1.14×**
xz	failed	failed	failed	**1.25×**
mcf	3.89×	1.49×	failed	**1.13×**
libquantum	5.55×	4.62×	failed	**1.12×**
hmmer	5.69×	3.03×	failed	**1.39×**
Zydis	21.34×	>100×	failed	**1.16×**
yyjson	25.36×	>100×	failed	**1.44×**

Table 6. Performance overhead of ExaGear and LLMSbt on Kunpeng 920 versus native AArch64 execution.

Testcases	ExaGear	LLMSbt	Testcases	ExaGear	LLMSbt
adpcm	45.36×	**1.12×**	loops	4.33×	**1.94×**
gsm	47.90×	**1.16×**	parser	2.50×	**1.09×**
mips	47.03×	**1.17×**	radix2	36.67×	**1.19×**
motion	47.98×	**1.22×**	zlib	24.54×	**1.10×**
aes	48.45×	**1.21×**	bzip2	**1.09×**	1.25×
sha1	44.11×	**1.24×**	xz	**1.10×**	1.23×
blowfish	41.11×	**1.27×**	mcf	1.21×	**1.13×**
jpeg	30.63×	**1.18×**	libquantum	**1.53×**	1.69×
cjpeg	4.80×	**1.37×**	hmmer	**1.20×**	1.24×
sha256	46.88×	**1.20×**	Zydisn	37.90×	**1.08×**
linpack	2.30×	**1.19×**	yyjson	28.43×	**1.59×**
nnet	**2.13×**	2.44×			

binary-heap, and hash-table. Using the perf profiling tool on the rk3588 platform, Table 7 presents the number of executed instructions and CPU cycles.

Table 7. Evaluation results of three algorithms. N and T represent the native binary and translated binary, respectively.

Algorithm	Source LoC	#Executed Insts (N / T)	#CPU Cycles (N / T)	Performance
avl-tree	1.2K	2,327M / 3,526M	1,016M / 1,292M	1.20×
binary-heap	0.5K	143.7M / 208M	120.4M / 120.8M	1.02×
hash-table	1.1K	353.3M / 477.4M	210M / 225M	1.12×

As shown in Table 7, translated binaries (marked T) consistently execute more instructions and consume more CPU cycles than native binaries (marked N). However, the increase in cycles is not always proportional to the increase in instructions. For instance, while the translated avl-tree executes 50% more instructions, it consumes only 27% more CPU cycles. This discrepancy suggests that performance is heavily influenced by microarchitectural features, such as pipelining, branch prediction, and caching. The instruction sequence and memory access patterns in translated binaries differ considerably from those in compiler-generated native code, which can substantially affect execution speed. Our evaluation indicates that the CPU pipeline effectively mitigates much of the overhead from increased instruction count.

Table 8. Detailed performance profiling with perf. The data is collected on the rk3588 device.

Algorithm	Branch Misses (N / T)	Cache Misses (N / T)	Stalled Cycles Frontend (N / T)	Stalled Cycles Backend (N / T)
avl-tree	2.3M / 2.1M	12M / 12.2M	123.4M / 10.6M	31.8M / 131.6M
binary-heap	135K / 126.6K	32K / 34K	332K / 334K	28.5M / 25M
hash-table	61.2K / 56K	299K / 316K	1.8M / 1.9M	42M / 41.7M

For a deeper insight, we used perf to record microarchitecture level metrics, with the results shown in Table 8. The data reveals a key finding: native binaries generally exhibit more branch misses, whereas translated binaries suffer from more cache misses. This is likely because the repetitive, patterned code resulting from translation rules is more friendly to branch predictors. However, the translation process does not account for memory layout, leading to lower cache efficiency. The avl-tree is a clear example, with a significant increase in cache misses (over 200K more), which results in a high number of backend stalled cycles (notably, one cache miss can cause a long stall). This indicates that avl-tree's overhead stems primarily from poor cache efficiency.

Overall, executing more instructions does not necessarily lead to high overhead. Performance depends heavily on how well the translated code utilizes CPU microarchitecture. Designing a binary translator that effectively accounts for the microarchitecture remains an open challenge, and investigating the intricate relationship between binary code and CPU pipeline behavior is an intriguing topic for future research.

7 Discussion

Remaining Challenges. While this work establishes the viability of a novel, LLM-based static translation pipeline, several orthogonal challenges remain. These long-standing issues represent important directions for future work, each requiring substantial, dedicated research and engineering efforts beyond the scope of a single paper.

1) Concurrent Programs. Translating concurrent programs is challenging due to the different memory models between architectures. Nevertheless, prior research offers a solid foundation by inserting and optimizing fences in lifted IR code [16,

34, 46, 91]. Although these methods do not focus on assembly-level translation, future research could adapt them to our static assembly-to-assembly translation paradigm.

2) C++ Programs. C++ features like exceptions and virtual functions, which rely on compiler-specific implementations, present unique hurdles for binary rewriting. These features complicate CFGs and data flow, making disassembly more difficult. Since our method relies on correct disassembly code, future research could focus on improving existing binary rewriting and disassembly techniques to cover these complex language features.

3) Floating-Point Operations. Architectural differences complicate the translation of floating-point operations. The x86-64 uses an eight-register stack for its x87 Floating Point Unit (FPU) [66], where AArch64's Advanced SIMD (NEON) [9, 68] lacks a floating-point stack. Translating these hardware-dependent instructions demands significant manual effort and expertise that is beyond this paper's scope. Current solutions often rely on carefully crafted helper functions. Due to this complexity, we exclude these instructions from our implementation, leaving a more efficient automated method as future work.

4) System Calls. Our evaluation targets user-level applications where dynamically linked native libraries handle system calls, simplifying translation since library APIs are consistent across architectures. However, translating system calls is inherently complex due to differences in conventions and data structure layouts. Our preliminary findings reveal inconsistencies in binary-level data structures between Linux kernels of different architectures (more details in the Appendix A.2). Initial experiments suggest these issues can be resolved by patching native libraries to enforce data structure consistency, with minimal performance impact. Future research could explore automated methods to identify and resolve such issues.

Supplementing with Manually-Crafted Rules. Although LLMs show significant potential in binary translation, they can struggle with complex instructions due to reasoning limitations. While our symbolic verification method automatically detects such failures, fixing them may require human intervention. In our experiments, we manually created 68 translation rules for cases in which LLM failed.

The practicality of this manual approach is supported by our methodology, which intentionally keeps translation code snippets short and their data flow simple. This design allows an experienced developer, referencing the relevant ARM documentation, to author an equivalent AArch64 code for a given snippet in minutes without extensive coding. While this requires expertise, our automatic semantic verification drastically reduces the manual validation effort, accelerating the entire workflow. Once verified, the translated code is parameterized into a reusable rule.

8 Related Work

Translation Validation. Ensuring the correctness of code transformations is a critical challenge. Recent efforts [30] have focused on validating the process of lifting binaries to LLVM IR by employing scalable instruction-level checking and formal verification to guarantee semantic equivalence. Within the compiler toolchain, tools like Alive2 [75] provide bounded translation validation to find bugs in LLVM's optimization passes. Other techniques, such as semantic program alignment [23], verify equivalence by matching program states along their execution traces based on semantic information, a method particularly effective for complex transformations that significantly restructure code. Complementing these, language-parametric approaches [62] offer more generalized validation frameworks that are not tied to a specific transformation, further strengthening the reliability of modern compilers. Nevertheless, these works primarily target high-level IR transformations, whereas our work is centered on assembly-level cross-architecture translation.

Extensions in Binary Translation. Recent advancements in binary translation have focused on addressing the complexities of modern software, particularly in areas like concurrency and post-compilation optimization. The challenge of translating for weak memory model architectures has been tackled by both static approaches, like Lasagne [91], and dynamic systems such as Risotto [46]. For multithreaded applications, hybrid recompilation techniques like Polynima [34] have emerged to balance performance and translation overhead. Researchers have also developed solutions for specific technical hurdles that impact correctness and efficiency. For instance, WYTIWYG [83] presents a method for dynamic stack-layout recovery, which enables post-compilation optimization for binaries, while Tiaozhuan [70] introduces a general optimization for indirect branches, a common performance bottleneck. Another promising direction involves leveraging information beyond the raw binary. Plankton [119], for example, supports full-fledged static analysis on lifted IR by utilizing debug information, while Biotite [22] uses source-level knowledge to generate higher-performance code. In the broader context of binary lifting, adjacent research explores decompilation to recover high-level source code (e.g., C) from binaries. Although decompilation is inherently challenging and often error-prone [73], recent advancements utilizing LLMs have substantially improved decompilation fidelity for small-scale programs [99, 111, 120]. They are, however, orthogonal to our work, which focuses on binary-level translation that produces efficient code and supports scalable verification.

Automated Generation of Translation Logic. A significant line of research has focused on automating the creation of translation logic to reduce manual effort and improve the quality of generated code. Early work in this area utilized

peephole superoptimizers [14] to automatically discover optimal, short instruction sequences for translation. Other research has improved the front-end of the translation process, presenting novel compiler-leveraged techniques for lifting assembly to a high-level intermediate representation [50]. More recently, this has evolved into a broader learning-based paradigm. For instance, several studies have shown how translation rules for dynamic binary translators can be automatically learned from compiler-generated binary code, an approach demonstrated by Wang et al. [105] and Song et al. [97]. This learning-based methodology has been successfully extended to the more demanding context of system-level translation [61] and has been made more data-efficient through techniques like parameterization [60], which allows deriving more general rules from less training data. In contrast, our method leverages LLMs to directly generate high-quality assembly code, eliminating the reliance on extensive training data or a separate compilation process.

9 Conclusion

This paper presents a novel static binary translation paradigm that converts x86-64 binaries to AArch64, producing portable binaries with near native performance. By utilizing LLMs, our approach substantially reduces the manual effort and expertise required to build translation engines. We ensure translation correctness through efficient semantic verification of small, simplified code snippets. Our research prototype, LLMSbt, has been shown to outperform both traditional open-source tools and a leading industrial translator. This work signals a promising new avenue for research in binary translation techniques.

Acknowledgments

This work was supported in part by the Hong Kong RGC Postdoctoral Fellowship Scheme (PDFS2324-6S08), the HKUST Bridge-the-Gap Fund (BGF.001.2025), and the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118). We also thank the HKUST Fok Ying Tung Research Institute and the National Supercomputing Center in Guangzhou (Nansha Sub-center) for providing computational resources, and the Collaborative Innovation Center of Novel Software Technology and Industrialization (Jiangsu, China) for their support.

References

[1] Useful resources for developing with the rk3588. https://github.com/choushunn/awesome-RK3588/tree/develop, 2024.

[2] Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman, and Donald E Porter. X86-64 instruction usage among c/c++ applications. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 68–79, 2019.

[3] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, et al. Binrec: dynamic binary lifting and recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[4] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An {In-Depth} analysis of disassembly on {Full-Scale} x86/x64 binaries. In *25th USENIX security symposium (USENIX security 16)*, pages 583–600, 2016.

[5] Apple. Apple introduces m4 chip. https://www.apple.com/ci/newsroom/2024/05/apple-introduces-m4-chip/, 2024.

[6] Apple. Apple silicon. https://developer.apple.com/documentation/apple-silicon, 2024.

[7] Apple. Porting your macos apps to apple silicon. https://developer.apple.com/documentation/apple-silicon/porting-your-macos-apps-to-apple-silicon, 2024.

[8] Apple. Rosetta translation environment. https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment, 2024.

[9] ARM. What is neon? https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/What-is-NEON-, 2024.

[10] ARM. Application binary interface for the arm® architecture – the base standard. https://github.com/ARM-software/abi-aa/blob/2982a9f3b512a5bfdc9e3fea5d3b298f9165c36b/bsabi32/bsabi32.rst, 2025.

[11] ARM. Procedure call standard for the arm architecture. https://github.com/ARM-software/abi-aa/blob/2982a9f3b512a5bfdc9e3fea5d3b298f9165c36b/aapcs32/aapcs32.rst#the-base-procedure-call-standard, 2025.

[12] Avast. Retargetable machine-code decompiler based on llvm. https://github.com/avast/retdec, 2025.

[13] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[14] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *OSDI*, volume 8, pages 177–192, 2008.

[15] Daniel Bartholomew. Qemu: a multihost, multitarget emulator. *Linux Journal*, 2006(145):3, 2006.

[16] Martin Beck, Koustubha Bhat, Lazar Stričević, Geng Chen, Diogo Behrens, Ming Fu, Viktor Vafeiadis, Haibo Chen, and Hermann Härtig. Atomig: automatically migrating millions lines of code from tso to wmm. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 61–73, 2023.

[17] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. California, USA, 2005.

[18] Jacob Bramley. Using the stack in aarch64: Implementing push and pop. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/using-the-stack-in-aarch64-implementing-push-and-pop, 2015.

[19] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018.

[20] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[21] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45, 2024.

[22] Changbin Chen, Shu Sugita, Yotaro Nada, Hidetsugu Irie, Shuichi Sakai, and Ryota Shioya. Biotite: A high-performance static binary translator using source-level information. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, pages 167–179, 2025.

[23] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1027–1040, 2019.

[24] Jamieson M Cobleigh, Dimitra Giannakopoulou, and Corina S Păsăreanu. Learning assumptions for compositional verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346. Springer, 2003.

[25] Florin Cuconasu, Giovanni Trappolini, Federico Siciliano, Simone Filice, Cesare Campagnano, Yoelle Maarek, Nicola Tonellotto, and Fabrizio Silvestri. The power of noise: Redefining retrieval for rag systems. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 719–729, 2024.

[26] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*, 2024.

[27] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, 1989.

[28] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[29] Amanieu d'Antras, Cosmin Gorgovan, Jim Garside, John Goodacre, and Mikel Luján. Hypermambo-x64: Using virtualization to support high-performance transparent binary translation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 228–241, 2017.

[30] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S Adve, and Christopher W Fletcher. Scalable validation of binary lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 655–671, 2020.

[31] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S Adve, and Grigore Roşu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1133–1148, 2019.

[32] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[33] B De Sutter, K De Bosschere, Peter Keyngnaert, and Bart Demoen. On the static analysis of indirect control transfers in binaries. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 2, pages 1013–1019. CSREA PRESS, 2000.

[34] Chinmay Deshpande, Fabian Parzefall, Felicitas Hetzelt, and Michael Franz. Polynima: Practical hybrid recompilation for multithreaded binaries. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1126–1141, 2024.

[35] Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. rev. ng: A multi-architecture framework for reverse engineering and vulnerability discovery. In *2018 International Carnahan Conference on Security Technology (ICCST)*, pages 1–5. IEEE, 2018.

[36] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. rev. ng: a unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 131–141, 2017.

[37] Artem Dinaburg and Andrew Ruef. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[38] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511. IEEE, 2020.

[39] EEMBC. Coremark-pro. https://www.eembc.org/coremark/, 2025.

[40] Alexis Engelke, Dominik Okwieka, and Martin Schulz. Efficient llvm-based dynamic binary translation. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 165–171, 2021.

[41] Alexis Engelke and Martin Schulz. Instrew: Leveraging llvm for high performance dynamic binary instrumentation. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 172–184, 2020.

[42] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1075–1092, 2020.

[43] Andrei Frumusanu. Amazon's arm-based graviton2 against amd and intel: Comparing cloud compute, 2020.

[44] Shay Gal-On and Markus Levy. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 6(23):87, 2012.

[45] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2(1), 2023.

[46] Redha Gouicem, Dennis Sprokholt, Jasper Ruehl, Rodrigo CO Rocha, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. Risotto: A dynamic binary translator for weak memory model architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 107–122, 2022.

[47] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: system programming guide, Part*, 2(11):0–40, 2011.

[48] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[49] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1192–1195. IEEE, 2008.

[50] Niranjan Hasabnis and R Sekar. Lifting assembly to intermediate representation: A novel approach leveraging compilers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 311–324, 2016.

[51] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, 2018.

[52] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[53] Hex-Rays. Ida pro powerful disassembler, decompiler and a versatile debugger. https://hex-rays.com/ida-pro, 2025.

[54] HiSilicon. Kunpeng chipsets. https://www.hisilicon.com/en/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920, 2024.

[55] R. Nigel Horspool and Nenad Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[56] Simon Howard. A library of common data structures and algorithms written in c. https://github.com/fragglet/c-algorithms, 2024.

[57] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. A survey on hallucination in large language

models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 2023.

[58] Huawei. Huawei kunpeng exagear. https://www.hikunpeng.com/document/detail/en/kunpengdevps/ug-exagear/usermanual/kunpengexagear_06_0003.html, 2024.

[59] ibireme. The fastest json library in c. https://github.com/ibireme/yyjson, 2024.

[60] Jinhu Jiang, Rongchao Dong, Zhongjun Zhou, Changheng Song, Wenwen Wang, Pen-Chung Yew, and Weihua Zhang. More with less–deriving more translation rules with less training data for dbts using parameterization. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 415–426. IEEE, 2020.

[61] Jinhu Jiang, Chaoyi Liang, Rongchao Dong, Zhaohui Yang, Zhongjun Zhou, Wenwen Wang, Pen-Chung Yew, and Weihua Zhang. A system-level dynamic binary translator using automatically-learned translation rules. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 423–434. IEEE, 2024.

[62] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S Adve, and Grigore Roşu. Language-parametric compiler validation with application to llvm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1004–1019, 2021.

[63] Hyungseok Kim, Soomin Kim, and Sang Kil Cha. Towards sound reassembly of modern x86-64 binaries. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1317–1333, 2025.

[64] Hyungseok Kim, Soomin Kim, Junoh Lee, Kangkook Jee, and Sang Kil Cha. Reassembly is hard: a reflection on challenges and strategies. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1469–1486, 2023.

[65] Johannes Kinder. Static analysis of x86 executables. 2010.

[66] Daniel Kusswurm and Daniel Kusswurm. X87 fpu programming. *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX*, pages 103–131, 2014.

[67] Celine Lee, Abdulrahman Mahmoud, Michal Kurek, Simone Campanoni, David Brooks, Stephen Chong, Gu-Yeon Wei, and Alexander M Rush. Guess & sketch: Language model guided transpilation. *arXiv preprint arXiv:2309.14396*, 2023.

[68] Sung-Jin Lee, Sang-Soo Park, and Ki-Seok Chung. Efficient simd implementation for accelerating convolutional neural network. In *Proceedings of the 4th International Conference on Communication and Information Processing*, pages 174–179, 2018.

[69] Xinyu Li, Guangyao Guo, Yanzhi Lan, Feng Xue, Chenji Han, Gen Niu, and Fuxin Zhang. Tiaozhuan: A general and efficient indirect branch optimization for binary translation. *ACM Transactions on Architecture and Code Optimization*, 2024.

[70] Xinyu Li, Guangyao Guo, Yanzhi Lan, Feng Xue, Chenji Han, Gen Niu, and Fuxin Zhang. Tiaozhuan: A general and efficient indirect branch optimization for binary translation. *ACM Transactions on Architecture and Code Optimization*, 22(1):1–26, 2025.

[71] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with smt solvers. *ACM SIGPLAN Notices*, 49(1):607–618, 2014.

[72] Zongjie Li, Chaozheng Wang, Pingchuan Ma, Chaowei Liu, Shuai Wang, Daoyuan Wu, Cuiyun Gao, and Yang Liu. On extracting specialized code abilities from large language models: A feasibility study. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[73] Zhibo Liu and Shuai Wang. How far we have come: Testing decompilation correctness of c decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 475–487, 2020.

[74] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. Sok: Demystifying binary lifters through the lens of downstream applications. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1100–1119. IEEE, 2022.

[75] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79, 2021.

[76] William Mahoney and J Todd McDonald. Enumerating x86-64 - it's not as easy as counting. In *Technical report, University of Nebraska Omaha*, 2021.

[77] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. The convergence of source code and binary vulnerability discovery–a case study. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 602–615, 2022.

[78] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2099–2116, 2022.

[79] Xiaozhu Meng and Barton P Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35, 2016.

[80] Microsoft. How emulation works on arm. https://learn.microsoft.com/en-us/windows/arm/apps-on-arm-x86-emulation, 2024.

[81] Trail of Bits. Framework for lifting x86, amd64, aarch64, sparc32, and sparc64 program binaries to llvm bitcode. https://github.com/lifting-bits/mcsema, 2025.

[82] OpenAI. Gpt-4 is openai's most advanced system, producing safer and more useful responses. https://openai.com/index/gpt-4/, 2024.

[83] Fabian Parzefall, Chinmay Deshpande, Felicitas Hetzelt, and Michael Franz. What you trace is what you get: dynamic stack-layout recovery for binary recompilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1250–1263, 2024.

[84] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. Xda: Accurate, robust disassembly with transfer learning. *arXiv preprint arXiv:2010.00770*, 2020.

[85] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with {SymCC}: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, 2020.

[86] Sebastian Poeplau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *NDSS 2021, Network and Distributed System Security Symposium*. Internet Society, 2021.

[87] QEMU. emit.c.inc. https://github.com/qemu/qemu/blob/bc6afa1c711da5b4f37c9685a812c77b114d84cb/target/i386/tcg/emit.c.inc#L2468, 2024.

[88] Zhenxiao Qi, Jie Hu, Zhaoqi Xiao, and Heng Yin. {SymFit}: Making the common (concrete) case fast for {Binary-Code} concolic execution. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 415–432, 2024.

[89] Qualcomm. Snapdragon x elite. https://www.qualcomm.com/products/mobile/snapdragon/laptops-and-tablets/snapdragon-x-elite, 2024.

[90] NGUYEN Anh Quynh and DANG Hoang Vu. Unicorn: Next generation cpu emulator framework. *BlackHat USA*, 476, 2015.

[91] Rodrigo CO Rocha, Dennis Sprokholt, Martin Fink, Redha Gouicem, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. Lasagne: A static binary translator for weak memory model architectures. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 888–902,

2022.

[92] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[93] Agam Shah. We're closing the gap with arm and x86, claims sifive: New risc-v cpu core for pcs, servers, mobile incoming ś the register, 2021.

[94] Ken Shirriff. Silicon reverse-engineering: the intel 8086 processor's flag circuitry. https://www.righto.com/2023/02/silicon-reverse-engineering-intel-8086.html, 2023.

[95] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*, pages 138–157. IEEE, 2016.

[96] Douglas R Smith. The design of divide and conquer algorithms. *Science of Computer Programming*, 5:37–58, 1985.

[97] Changheng Song, Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Weihua Zhang. Unleashing the power of learning: An enhanced {Learning-Based} approach for dynamic binary translation. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 77–90, 2019.

[98] Tom Spink, Harry Wagstaff, and Björn Franke. A retargetable system-level dbt hypervisor. *ACM Transactions on Computer Systems (TOCS)*, 36(4):1–24, 2020.

[99] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3473–3487, 2024.

[100] Rev.ng team. Failure when trying to translate a simple dynamic x86-64 binary compiled with clang. https://github.com/revng/revng/issues/317#issuecomment-1721136071, 2025.

[101] Michael James Van Emmerik. *Static single assignment for decompilation*. University of Queensland, 2007.

[102] Fish Wang and Yan Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[103] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[104] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, 2015.

[105] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Enhancing cross-isa dbt through automatically learned translation rules. *ACM SIGPLAN Notices*, 53(2):84–97, 2018.

[106] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. A general persistent code caching framework for dynamic binary translation ({{{{{DBT}}}}}). In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 591–603, 2016.

[107] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. Coremu: a scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 213–222, 2011.

[108] Albert Webson and Ellie Pavlick. Do prompt-based models really understand the meaning of their prompts? *arXiv preprint arXiv:2109.01247*, 2021.

[109] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[110] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 133–147, 2020.

[111] Wai Kin Wong, Daoyuan Wu, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Decllm: Llm-augmented recompilable decompilation for enabling programmatic use of decompiled code. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1841–1864, 2025.

[112] Benyi Xie, Yue Yan, Chenghao Yan, Sicheng Tao, Zhuangzhuang Zhang, Xinyu Li, Yanzhi Lan, Xiang Wu, Tianyi Liu, Tingting Zhang, et al. An instruction inflation analyzing framework for dynamic binary translators. *ACM Transactions on Architecture and Code Optimization*, 21(2):1–25, 2024.

[113] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4554–4568, 2024.

[114] S Bharadwaj Yadavalli and Aaron Smith. Raising binaries to llvm ir with mctoll (wip paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 213–218, 2019.

[115] Yapeng Ye, Zhuo Zhang, Qingkai Shi, Yousra Aafer, and Xiangyu Zhang. D-arm: Disassembling arm binaries by lightweight superset instruction interpretation and graph modeling. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2391–2408. IEEE, 2023.

[116] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. {DeepDi}: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2709–2725, 2022.

[117] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. Siren's song in the ai ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219*, 2023.

[118] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[119] Anshunkang Zhou, Chengfeng Ye, Heqing Huang, Yuandao Cai, and Charles Zhang. Plankton: Reconciling binary code and debug information. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 912–928, 2024.

[120] Zhiping Zhou, Xiaohong Li, Ruitao Feng, Yao Zhang, Yuekang Li, Wenbu Feng, Yunqian Wang, and Yuqing Li. Fidelitygpt: Correcting decompilation distortions with retrieval augmented generation. In *Network and Distributed System Security (NDSS) Symposium 2026*.

[121] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 21–29, 2022.

[122] zyantific. Fast and lightweight x86/x86-64 disassembler and code generation library. https://github.com/zyantific/zydis, 2024.

A Appendix

Table 9. Evaluated software.

Name	Description
adpcm	Adaptive differential pulse code modulation decoder and encoder
gsm	Linear predictive coding analysis of audio signal
mips	Simplified MIPS processor
motion	Motion vector decoding for MPEG-2
aes	Advanced encryption standard
sha1	Secure hash algorithm
blowfish	Symmetric-key block cipher
jpeg	Image decompression
cjpeg	JPEG image compression
sha256	Secure hash algorithm
linpack	Gaussian elimination with partial pivoting
nnet	Neural net
loops	Kernels based on Livermore loops
parser	XML parsing
radix2	Radix-2 fast Fourier transform
zlib	ZIP compression
bzip2	File compression
xz	File compressions
mcf	Single-depot vehicle scheduling
libquantum	Physics / Quantum computing
hmmer	Gene sequence database searching
zydis	Disassembler library
yyjson	JSON library

A.1 Performance of Static Concise IR Lifters

We evaluated three state-of-the-art static concise IR lifters, including Mctoll, RetDec, and Rev.ng, on the same set of 23 software used in our evaluation. We found that these lifters failed in all cases. Specifically, RetDec and Rev.ng can produce lifted IR code for all cases, but the lifted IR code is either non-compilable or functionally incorrect. RetDec extensively skips and leaves unimplemented instructions in its output undefined, while Rev.ng translated binaries crash immediately upon execution. On the other hand, Mctoll fails to lift any of the cases, with error messages reported in Table 10.

A.2 Binary-Level Data Structure Inconsistency

As shown in Fig. 10, Linux kernel defines the `stat` structure in different ways for x86-64 [1] and AArch64 [2] architectures. Specifically, the order of `st_nlink` and `st_mode` are different. And `glibc` handles this difference by explicitly parsing data returned by system calls according to the target architecture [3], but still retains such binary-level data structure inconsistency. In this example, `glibc` keep the same API for `fstat` system call. Source-level references to specific members in `stat` structure are still valid. However, since binary-level references are achieved with constant offsets, the inconsistency in the layout of `stat` structure may cause the translated binary to access wrong members.

[1] https://github.com/torvalds/linux/blob/8cf0b93919e13d1e8d4466eb4080a4c4d9d66d7b/arch/x86/include/uapi/asm/stat.h#L42

[2] https://github.com/torvalds/linux/blob/8cf0b93919e13d1e8d4466eb-4080a4c4d9d66d7b/arch/arm/include/uapi/asm/stat.h#L57

[3] https://github.com/lattera/glibc/blob/895ef79e04a953cac1493863bcae29ad85657ee1/sysdeps/unix/sysv/linux/x86/bits/stat.h#L119

Table 10. Mctoll's Error Messages.

Test	Error Message
adpcm	Branch instruction terminating basic block computing jump table
gsm	Unhandled memory referencing instruction
mips	Branch instruction terminating basic block computing jump table
motion	Unhandled PUSH instruction with non-AddrRegFrm source operand
aes	Null reaching definition found during reaching definition fixup
sha1	Alignment is neither 0 nor a power of 2
blowfish	Null reaching definition found during reaching definition fixup
jpeg	Branch instruction terminating basic block computing jump table
cjpeg	Unknown opcode
sha256	Unknown opcode
linpack	Unknown opcode
nnet	Unknown opcode
loops	Unknown opcode
parser	Unknown opcode
radix2	Unknown opcode
zlib	Unknown opcode
bzip2	cast<Ty>() argument of incompatible type!
xz	Unexpected block terminator found
mcf	Unknown opcode
libquantum	Unhandled SSE Packed Single Prefix encountered
hmmer	Branch instruction terminating basic block computing jump table
zydis	Unexpected number of successors of switch block
yyjson	Unexpected block terminator found

```
struct stat64 {
    __dev_t st_dev;
    __ino64_t st_ino;
    __nlink_t st_nlink;
    __mode_t st_mode;
    __uid_t st_uid;
    __gid_t st_gid;
    ...
};
```

In Linux kernel for x86-64

```
struct stat64 {
    __dev_t st_dev;
    __ino64_t st_ino;
    __mode_t st_mode;
    __nlink_t st_nlink;
    __uid_t st_uid;
    __gid_t st_gid;
    ...
};
```

In Linux kernel for AArch64

```
struct stat64 {
    __dev_t st_dev;         /* Device.  */
# ifdef __x86_64__
    __ino64_t st_ino;       /* File serial number.  */
    __nlink_t st_nlink;     /* Link count.  */
    __mode_t st_mode;       /* File mode.  */
# else
    unsigned int __pad1;
    __ino_t __st_ino;       /* 32bit file serial number.*/
    __mode_t st_mode;       /* File mode.  */
    __nlink_t st_nlink;     /* Link count.  */
# endif
...
};
```

Relevant code in `glibc`.

Figure 10. Binary-level data structure inconsistency.