Siro: Empowering Version Compatibility in Intermediate Representations via Program Synthesis

Bowen Zhang
bzhangbr@cse.ust.hk
The Hong Kong University of Science and Technology
China

Wei Chen*
wchenbt@cse.ust.hk
The Hong Kong University of Science and Technology
China

Peisen Yao
pyaoaa@zju.edu.cn
Zhejiang University
China

Chengpeng Wang
cwangch@cse.ust.hk
The Hong Kong University of Science and Technology
China

Wensheng Tang
wtangae@cse.ust.hk
The Hong Kong University of Science and Technology
China

Charles Zhang
charlesz@cse.ust.hk
The Hong Kong University of Science and Technology
China

Abstract

This paper presents Siro, a new program transformation framework that translates between different versions of Intermediate Representations (IR), aiming to better address the issue of IR version incompatibility on IR-based software, such as static analyzers. We introduce a generic algorithm skeleton for Siro based on the divide-and-conquer principle. To minimize labor-intensive tasks of the implementation process, we further employ program synthesis to automatically generate translators for IR instructions within vast search spaces. Siro is instantiated on LLVM IR and has effectively helped to produce ten well-functioning IR translators for different version pairs, each taking less than three hours. From a practical perspective, we utilize these translators to assist static analyzers and fuzzers in reporting bugs and achieving accuracy of 91% and 95%, respectively. Remarkably, Siro has already been deployed in real-world scenarios and makes existing static analyzers available to safeguard the Linux kernel by uncovering 80 new vulnerabilities.

ACM Reference Format:
Bowen Zhang, Wei Chen, Peisen Yao, Chengpeng Wang, Wensheng Tang, and Charles Zhang. 2024. Siro: Empowering Version Compatibility in Intermediate Representations via Program Synthesis. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3620666.3651366

*Corresponding author.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04…$15.00
https://doi.org/10.1145/3620666.3651366

1 Introduction

Program analysis involves automatically analyzing program properties through static and dynamic approaches. Central to this technique is the use of Intermediate Representations (IR), which are mid-level program representations produced by compilers [3]. From a behavioral perspective, program analyses can be abstracted into various IR manipulations. For example, static analyzers [4, 82, 84, 89] traverse the IR to reason about program facts, symbolic executors [9] symbolically interpret the IR to compute path conditions, and fuzzers [1, 27] perform instrumentation at the IR level to obtain runtime feedback. Due to the close association with IR, we refer to their implementations as *IR-based software*.

When the compilers frequently update themselves, compatibility issues arise between different versions of IR. Consequently, IR-based software, which is typically built upon a specific version of IR, cannot accept higher versions of IR programs as input. This version capability issue also hinders the collaboration of different IR-based software targeting different IR versions, such as integrating an existing static analyzer into a white-box fuzzer. Existing approaches to address this issue involve either refactoring the IR-based software to higher versions or compiling programs to the version supported by the software. Both approaches require significant manual work, which imposes a heavy burden on program analysis researchers and practitioners.

A possible workaround is to introduce an "*IR translator*" that converts IR programs between versions, allowing the translated IR programs to be directly accepted by IR-based software. Unfortunately, previous attempts at implementing similar proposals within the LLVM community have been unsuccessful, indicating a heavy reliance on manual labor [58, 63].

To enhance IR version compatibility, we introduce Siro (Synthesis-powered IR translation), a framework for efficiently implementing IR translators between different versions. The motivation is that while IR compatibility involves many changes, the information provided by IR for program analysis and the functionality of IR libraries remain immutable. Specifically, Siro alleviates manual efforts through three key aspects. First, it leverages the IR library within the compiler as readily available material to construct IR translators. Next, Siro divides and conquers the hierarchical structure of the IR by creating a version-agnostic algorithm skeleton for IR translation. Within this skeleton, the remaining task is to translate various IR instructions. Furthermore, recognizing the significant presence of common instructions across IR versions, we transform the writing of their translators into a test case-guided program synthesis problem. Specifically, our synthesis system takes user-provided IR programs and employs their execution behavior as an oracle to incrementally achieve the correct implementation of these instruction translators. As a result, our framework replaces the manual effort of writing IR translators by providing test cases and only handling a small number of new instructions.

The experimental results are highly encouraging. Siro succeeded in synthesizing ten LLVM IR translators for different version pairs, requiring only ten days effort from one person. We used the generated IR translators to translate programs from both real-world projects and benchmark datasets. The evaluation resulted in accuracy rates of 91% for bug detection and 95% for fuzzing target reproduction, providing strong evidence for the feasibility of this technology. Actually, Siro has already been deployed in the industry, assisting a kernel bug detector in obtaining the IR program of the Linux kernel, leading to the discovery of 80 new bugs.

Our high contributions are highlighted as follows:

- We identified a practical issue across the program analysis community, known as the IR version compatibility issue, and highlighted the shortcomings of existing solutions.
- We explored a novel notion, IR translation, and proposed the Siro framework for efficiently implementing IR translators. Within the framework, we introduced a generic algorithmic skeleton and designed a program synthesis system to efficiently fill in the missing IR instruction translators in the skeleton.
- We conducted a comprehensive evaluation to demonstrate the feasibility of this technique and showcased the effectiveness and efficiency of Siro.

2 Background and Motivation

In this section, we first discuss the IR version trap and three technical choices to overcome it. We then present IR translation as our choice and highlight the associated challenges.

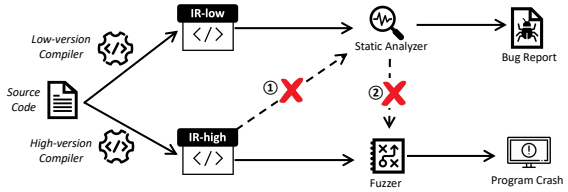

Figure 1. Scenarios of IR version trap

2.1 The IR Version Trap

Compilers evolve frequently. For instance, the popular compilers LLVM [60] and GCC [31] have undergone 20 and 47 version updates over the past decade, respectively. While backward compatibility is often limited [30, 44, 59], forward compatibility is not guaranteed. Ultimately, such compatibility issues render program analysis tools in lower versions unusable and hinder collaboration within the program analysis community. In this paper, we refer to this phenomenon as the "IR version trap." We illustrate the consequences of the IR version trap through two typical scenarios in Fig. 1.

Scenario I. (Static Analysis) A static bug detector was originally developed on top of a low-version IR. However, certain projects can only be compiled using a higher version of the compiler, generating IR programs in the higher version. Consequently, the static analyzer cannot accept these high versions of the IR programs as inputs, which is demonstrated by the edge ①, hampering the usability of the analyzer.

Scenario II. (White-box Fuzzing) Fuzzers [12, 13, 22] often rely on static analysis to obtain different semantic properties of the program under test, such as pointer information and numeric invariants. However, the fuzzers would not benefit from the static analyzers if they are developed on top of different IR versions, The edge ② shows the failure of the cooperation between two kinds of IR-based software.

Furthermore, the IR version trap hinders the development of the program analysis community, considering the increasing trend of combining static and dynamic analysis [42, 45, 48] or integrating multiple static analyses through layered design [10, 54, 67]. As the number of IR-based software increases, this trap becomes more prevalent and challenging to overcome. In what follows, we discuss several types of mitigations to this issue.

2.2 Escaping from IR Version Trap

The IR version trap has gained some attention in both academic and industrial practice. Next, we briefly introduce two common strategies and the new strategy we are exploring in Fig. 2. Initially, suppose the compiler can only generate a higher version IR, while the IR-based software S_{low} operates in a lower version, and thus, is inapplicable in higher version IR.

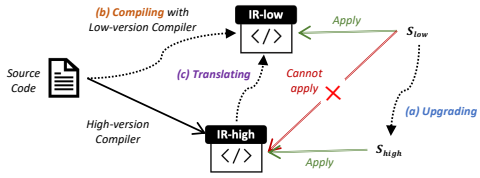

Figure 2. Three strategies to escape from the IR version trap

Table 1. Statistics of LLVM IR-based software

Software	Description	IR Version	# IR Versions	# Maintainers
KLEE	Symbolic execution engine	13.0	11	89
SeaHorn	Software model checker	5.0	2	19
SVF	Static value-flow analyzer	13.0	8	67
IKOS	Abstract interpretation framework	14.0	8	7

Strategy I: Upgrading. We show the upgrading approach in Fig. 2(a), which migrates an IR-based software to the high version (S_{high}), to be capable of taking the high version of IR as a valid input. However, upgrading the software against one new IR version is difficult and often requires laborious manual efforts since the developers have to deal with API changes and related compilation errors. In addition, since the upgrade nearly touches each component of IR-based software, all relevant developers have to review the functionality correctness. For example, a single LLVM version upgrade in the Rust compiler [78] takes over one month to finish, whose discussions involve 167 posts, 20 developers, and 33 commits. Such effort is needed whenever a new IR version is released. Similarly, as listed in Tab. 1, the upgrading approach also troubles developers in all other IR-based software. As a result, numerous IR-based software are unable to keep up with the latest IR versions, due to insufficient active maintainers. Therefore, it is impractical and laborious to address the IR version trap using the upgrading approach.

Strategy II: Compiling. Fig. 2(b) demonstrates the compiling approach that compiles a project with a low-version compiler. The generated IR is restricted to the low version and can be handled by the IR-based software directly. However, the complexity and diversity of build systems increase the difficulty of applying the approach. Specifically, the developers have to drown in the details of dependency management, compiling, and linking. Meanwhile, the weak compilation capability of low-version compilers can even worsen the situation, leading to failed compilations of many real-world projects. According to our investigation, open-source fundamental projects, including Linux, LLVM, QT, and Firefox [28, 56, 62, 73], cannot be compiled with low-version compilers. Therefore, the compiling approach fails to be a practical solution as well.

Strategy III: Translating. Another alternative is to translate the IR programs between different versions, which is inspired by the program transpilation technique [17]. As shown in Fig. 2(c), a high-version compiler first generates the IR in a high version, and then the IR translator downgrades the IR to low version, which can be further processed by the IR-based software. Obviously, the IR translation process is isolated from the IR-based software and the compilation process, which implies that it requires no modification to either the project's default compiler or the IR-based software itself. Additionally, while upgrading and compiling must be performed on a per-software basis, the IR translator offers a broad range of benefits that can be applied generally for multiple IR-based software. For example, both scenarios in Fig. 1 can be resolved with the translating approach.

Our Choice. After comparing the above strategies, it is evident that IR translation offers a promising prospect for addressing the IR version trap. It provides a flexible bridge, connecting different ecosystems caused by version discrepancies, thereby fostering interaction within the program analysis community. However, this approach has not been sufficiently discussed in the community, leading to uncertainties regarding its feasibility, algorithms, and efficient implementation. Through this article, we intend to address these concerns and enable the program analysis community to benefit from this technique.

3 IR Translation Essentials

This section reveals a holistic journey of designing the IR translation technique. It covers how we define the technical roadmap (§ 3.1), propose algorithmic frameworks (§ 3.2), and identify opportunities for program synthesis (§ 3.3).

3.1 Demystifying the IR Version Trap

The goal of IR translation is to help IR-based software resolve version traps and produce consistent analysis results. To achieve such an objective, we first formulate the IR from a program analysis perspective, then, we design the technical approach by examining the root cause of the IR version trap.

IR Formulation. As formulated in Fig. 3, IR provides program information to IR-based software in a hierarchical way. Specifically, A top-level IR program P consists of global variables G and functions F. The basic blocks B within each function form the program's control flow. The instructions I that are sequentially contained within each basic block serve as the fundamental semantic components within the IR, with each ***kind*** of instruction enforcing a specific operation on its operand values. Notably, instructions can reference almost all IR elements through such operand relationships. For instance, consider a call instruction $v_{call} \leftarrow \texttt{call}(f_1, 10, arg_3)$, which invokes a callee function f_1 using the constant value 10 and the argument value arg_3 from its caller as parameters, and then produces a return value v_{call}. In this example, four operand values are involved. Thus, despite the clarity of the formulation, why is IR-based software unable to manipulate such information across versions?

Understanding Version Trap. An IR-based software typically needs to load the serialized IR program into memory

$$
\begin{aligned}
\text{IR Program } P &:= F^+\ G^+ \\
\text{Function } F &:= f(arg_1, \cdots, arg_n)\{(B)^+\} \\
\text{Basic Block } B &:= (I)^+ \quad \text{Argument } arg \in Arg \\
\text{Instruction } I &:= v_0 \leftarrow \oplus(v_1, \cdots, v_n) \\
\text{Operator } \oplus &:= \texttt{add} \mid \texttt{load} \mid \texttt{branch} \mid \texttt{ret} \mid \texttt{call} \mid \cdots \\
\text{Value } v &:= G \mid Arg \mid F \mid B \mid I \mid C \\
\text{Global } g &\in G \quad \text{Constant } c \in C
\end{aligned}
$$

Figure 3. IR formulation

Table 2. IR libraries to construct an IR translator

IR Library	Functionality
IR Reader ❶	Load a persisted IR program into the memory.
IR Getter ❶	Access information from an IR memory object.
IR Builder ❷	Construct in-memory IR programs and IR elements.
IR Verifier ❷	Verify the integrity and legality of an IR program.
IR Writer ❷	Write an in-memory IR program into a persisted one.

and then conduct the analysis logic by using the APIs provided by the IR libraries to manipulate IR. During this process, the IR version trap can arise through three incompatibilities.

- **Text incompatibility.** The serialization format of IR programs is inconsistent between versions, resulting in incompatible IR loading.
- **API incompatibility.** The frequent changes in the APIs of IR libraries result in inconsistent implementations of the same analysis logic across different versions.
- **Semantic incompatibility.** New versions sometimes introduce new semantics through new IR instructions, necessitating special handling to ensure compatibility.

Notably, only the third one brings additions to our formulation, whereas the first two stem from engineering details.

Translation "In-memory". Facing such root causes, our philosophy is not to attempt to individually address the changes regarding the text format and API usages. Doing so would lead us into the same pitfalls as the compiling approach and the upgrading approach. On the contrary, we observe that regardless of how versions may change, the essential functionality provided by IR libraries remains immutable and robust. This fact can be leveraged to develop IR translators for various versions.

Tab. 2 lists the functionalities of IR libraries utilized to construct an arbitrary IR translator, where ❶ and ❷ represent the source and target IR versions, respectively. By flexibly combining two versions of IR libraries, we confine the IR translation tasks to the translation of ***in-memory*** IR programs, directly overcoming text incompatibility. Among the libraries, the IR getters and builders are the most important. The in-memory IR translation, at its core, utilizes these two kinds of libraries to implement a so-called "***Extract and Reconstruct***" principle: for any IR element, we first extract the information it provides using IR getters according to the formulation, and then use IR builders to reconstruct it in the target version. Readers will observe multiple instantiations of this principle within the translation algorithms.

> **Summary.** IRs are formulated to contain fundamental elements to describe programs. Despite the ***IR compatibility*** issue, the invariants of IR enable us to follow the "extract and reconstruct" principle to develop an ***in-memory*** approach with ***IR libraries***.

3.2 Translation Skeleton

Observing that instructions are the most crucial and complex components within IR, subject to notable instability in APIs and semantics, we break down an IR translator into two parts: a translation skeleton for IR elements except for instructions, and dedicated instruction translators. From a high level, the translation skeleton is version-agnostic and can be reused for different versions, and thus could be manually written with negligible yet one-time effort. On the contrary, instruction translators that are complex and heavily dependent to APIs of different versions, are more deserving to be automatically synthesized. In what follows, we introduce the skeleton.

As described in Alg. 1, the skeleton is built by ***dividing and conquering*** the in-memory, hierarchical structure of IR formulated in Fig. 3. We translate each IR element layer by layer in a top-down manner and encapsulate this into an interface, which facilitates program synthesis in the subsequent stages. Indeed, the extract and reconstruct principle permeates the entirety of the translation process. Specifically, the translation of the top-level IR program $P = (G, F)$, involves translating all global variables and functions. The translation of a function is divided into translating the argument list and its included basic blocks (`TranslateFunc`). For each basic block, we translate the instructions sequentially (`TranslateBlock`). Finally, to translate an instruction, we determine its specific instruction kind and dispatch it to corresponding instruction translation functions (`TranslateInst`). In the next subsection, we introduce how to construct these instruction translators.

> **Summary.** Version-independent translation logic could be extracted from IR translators to form a ***translation skeleton***, which is established following the ***divide-and-conquer*** principle.

3.3 Guidance to Instruction Translators

So far, through the in-memory approach and the version-independent skeleton, we have overcome text incompatibility and part of API incompatibility. To address the remaining compatibility problems, we divide instructions into "common" and "new" instructions and handle them in distinct

Algorithm 1: A high level skeleton of IR translation

Input: Source version IR program $P = (G, F)$;
Output: Target version IR program $P' = (G', F')$;
1 $G' \leftarrow \emptyset$; $F' \leftarrow \emptyset$;
2 **for** $g \in G$ **do** $G' \leftarrow G' \cup \{\texttt{TranslateGlobal}(g)\}$;
3 **for** $f = (Args, B) \in F$ **do**
4 $\quad F' \leftarrow F' \cup \{\texttt{TranslateFunc}(Args, B)\}$
5 **return** $P' = \texttt{CreateIRProgram}(G', F')$;
6 **Function** `TranslateFunc`$(Args, B)$**:**
7 $\quad Args' \leftarrow \emptyset$; $B' \leftarrow \emptyset$;
8 $\quad$ **for** $arg \in Args$ **do**
9 $\quad\quad Args' \leftarrow Args' \cup \{\texttt{TranslateArg}(arg)\}$;
10 $\quad$ **for** $b \in B$ **do** $B' \leftarrow B' \cup \{\texttt{TranslateBlock}(b)\}$;
11 $\quad$ **return** $f' = \texttt{CreateFunc}(Args', B')$
12 **Function** `TranslateBlock`(b)**:**
13 $\quad insts' \leftarrow \emptyset$;
14 $\quad$ **for** $i \in b$ **do** $insts' \leftarrow insts' \cup [\texttt{TranslateInst}(i)]$;
15 $\quad$ **return** $b' = \texttt{CreateBlock}(insts')$;
16 **Function** `TranslateInst`(i)**:**
17 $\quad$ **if** i *is* `add` **then** $i' \leftarrow \texttt{TranslateAdd}(i)$;
18 $\quad$ **else if** i *is* `load` **then** $i' \leftarrow \texttt{TranslateLoad}(i)$;
19 $\quad$ **else if** i *is* `branch` **then** $i' \leftarrow \texttt{TranslateBranch}(i)$;
20 $\quad$ **else if** i *is* ... **then** $i' \leftarrow \ldots$ ▷ others;
21 $\quad$ **return** i';

strategies. We also highlight the challenges of automatically synthesizing the instruction translators.

3.3.1 Handling Common Instructions. Common instructions refer to the instructions shared between two versions. Therefore, our "extract and construct" pattern is instantiated to a one-to-one mapping. However, implementing these common instructions remains challenging due to the requirement of developers to possess a deep understanding of the instructions. This complexity stems from two key factors:

- **Operands.** As evident in the formulation in Fig. 3, instructions involve various IR elements due to their complex operand relationships. Therefore, a correct translation necessitates a proper understanding of these relationships.
- **Sub-kinds.** Instructions of the same kind may implicitly have sub-kinds, resulting in divergent operand relationships and necessitating different translation logics.

We take the branch instruction as an example to illustrate such complexities in Fig. 4. Here, `TranslateBranch` is the desired implementation that translates instruction from the source version with type `Branch_s` to the target version with type `Branch_t`. In line 2, the check `inst.IsUncondBr()` is driven by the existence of sub-kinds, specifically determining if the branch instruction is conditional or not. As a result, the usage of APIs diverges to construct translation logic for different sub-kinds, i.e., lines 3-5 and 7-13. The translation process still enforces the "extract and reconstruct" manner,

```
1  Branch_t TranslateBranch(Branch_s inst) {
2    if(inst.IsUncondBr()) {
3      Block_s b = inst.GetBlock(0);
4      Block_t bb = TranslateBlock(b);
5      return Builder.CreateBr(bb);
6    } else {
7      Value_s cond = inst.GetCond();
8      Block_s b0 = inst.GetBlock(0);
9      Block_s b1 = inst.GetBlock(1);
10     Value_t condd = TranslateValue(cond);
11     Block_t bb0 = TranslateBlock(b0);
12     Block_t bb1 = TranslateBlock(b1);
13     return Builder.CreateCondBr(condd, bb0, bb1);
14   } }
```

Figure 4. Instruction translator of branch instruction

as we interweave the usage of IR getters and builders. Additionally, it is worth noting that the translation interfaces for each exposed IR element in the algorithm skeleton (as shown in Fig. 1) also play a significant role in this process. This integration becomes possible by adhering to the divide and conquer principle.

In fact, common instructions constitute the majority of the development effort in IR translators. Therefore, to spare developers from handling their complexities and dealing with unstable APIs, we choose to employ program synthesis to automatically generate their translators. Specifically, as shown in the following Def. 3.1, we abstract the desired translator for each common instruction kind k as a mapping $\mathbb{M}_k$, which serves as the goal of program synthesis.

Definition 3.1 (Common Instruction Translator)**.** For each common instruction kind $k \in K$, its instruction translator is defined as a mapping $\mathbb{M}_k : [\Sigma_k \mapsto \Lambda_k]$, where:

- $\sigma \in \Sigma_k$ represents a ***predicate*** to differentiate a sub-kind of k. It is an equality check on a specific property of the instruction, achieved via the IR getters of enum/bool type corresponding to instruction kind k. Additionally, $\mathbb{M}_k$ can include only one predicate *true*, indicating the presence of only one sub-kind. Otherwise, if $\mathbb{M}_k$ has multiple predicates, only one of them can be true at runtime.
- $\lambda \in \Lambda_k : I_k \mapsto I'_k$ denotes an ***atomic translator***. It handles the translation of a specific sub-kind of k, using three materials: IR getters of k, IR builders producing k, and operand translator interfaces. I_k and I'_k represent the sets of instructions of kind k in the source and target versions.

3.3.2 Handling New Instructions. New instructions can be introduced in higher versions, constituting a small proportion of the overall set of instructions. However, their translations are not suitable for program synthesis since new semantics typically cannot be fully equivalent to the combination of multiple common instructions. Therefore, we propose two principles: First, check the necessity of translation. Many new instructions can never be encountered by the

IR-based software, as they are designed for specific targets or serve as internal instructions used within optimizations e.g., Inc instruction in DEX IR [74]. By referring to documentation and community discussions, such unnecessary handling can be avoided. Second, achieve analysis preserving translation. Specifically, we can employ a one-to-many translation to ensure that the resulting IR programs are equivalent in terms of program analysis results [91]. Typically, ensuring that the translated instruction produces the same effects on control flow and data flow is sufficient.

3.3.3 Synthesis Challenges. So far, the IR translation technique itself has been discussed. Additionally, if the translators of common instructions in § 3.3.1 could be automatically generated through program synthesis, our system will be capable of efficiently producing IR translators for any version pair to conquer the IR version trap. However, applying program synthesis techniques to our problem is far from trivial. Since the program synthesizer does not understand IR translation and the APIs, our synthesis problem faces two challenges, including a large search space and unconventional validation.

Search Space. Synthesizing a translator for instructions utilizes three sets of APIs: IR getters, IR builders, and operand translators. Since instructions are crucial elements in IR, the getters/builders APIs are abundant in a compiler, which forms a vast number of possibilities. On the contrary, the right implementations are rare, not to mention the need to differentiate between various sub-kinds.

Validation. Worse, even if we generate some candidate translators for various instructions, we cannot independently validate their correctness. This is primarily because an IR program typically consists of multiple instructions. One can imagine that given an IR program as a test case, we would need to simultaneously verify multiple candidates. Such a combination further exacerbates the search space.

Summary. Common instructions translators are abstracted as ***atomic translators*** paired with ***predicates***. However, automatic synthesis encounters two challenges from ***search space*** and ***validation*** aspects.

4 Instruction Translator Synthesis

In this section, we provide an overview of our synthesis system (§ 4.1), followed by the introduction of three core aspects: generating candidates using type information (§ 4.2), refining candidates through test cases (§ 4.3), and optimizing the synthesis process to facilitate pruning (§ 4.4).

4.1 System Design

Fig. 5 illustrates Siro's synthesis system, which essentially is an iterative continuous search space reduction process for potential instruction translators. It takes IR libraries and test cases as initial inputs and ultimately produces a correct IR translator consisting of our translation skeleton and synthesized instruction translators. As such, the task of developers writing instruction translators could be replaced by simply collecting test cases, each of which is a simple IR program whose return value would serve as an oracle for us.

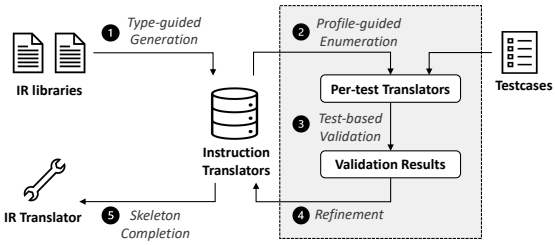

Figure 5. Overall synthesis system

Algorithm 2: Program synthesis process

Input: Test cases T. IR libraries Lib. Instruction kinds K.
Output: IR translator code.

```
1 Λ* ← Generate(Lib, K);                          ▷ ❶
2 M* ← ∅;
3 for t ∈ T do
4     τ_t ← Profile(t);                           ▷ ❷
5     PT_t ← Enumerate(Λ*, τ_t);                  ▷ ❷
6     PT_t^✓ ← Validate(PT_t, t);                 ▷ ❸
7     Refine(M*, PT_t^✓, τ_t);                    ▷ ❹
8 return CompleteSkeleton(M*);                    ▷ ❺
```

Specifically, Siro first utilizes the type information provided by IR libraries to generate numerous candidate atomic translators for each kind of instruction, whereas the correctness and sub-kind information of the candidates is initially unknown (❶). Next, we perform an iteration on the test cases to ***refine*** the candidates. Specifically, the test cases would help us continuously exercise the correctness of candidates while establishing the mapping between predicates and correct candidates (❷❸❹). After processing enough test cases, Siro generates instruction translators based on refined candidates, fills them into the translation skeleton in § 3.2, and produces a complete IR translator (❺).

The key factor behind the iteration is our "ad-hoc" design in Alg. 2, namely ***per-test translator***, which combines several candidate atomic translators and would only be used to translate the current test case. Specifically, to construct the per-test translators for test case t, Siro first probes its involved instructions with profiling technique, generating a table τ_t (line 4). Guided by τ_t, the candidate atomic translators Λ^* are enumerated to form the per-test translators PT_t (line 5). The purpose of these per-test translators is to translate case t and validate the oracle (line 6). Finally, from the succeeded ones $PT_{t^\checkmark}$, we can refine the mapping $\mathbb{M}^*$, which records an over-approximation of the correctness and paired predicates related to the candidates (line 7). In summary, each per-test translator can be likened to a multivariable

equation, where each variable represents an atomic translator. By constructing and solving more equations, we can further approximate the correct solution for each variable.

4.2 Type-guided Generation

The synthesis system starts by generating candidate atomic translators Λ^*. For each instruction kind k, the candidates Λ^*_k are the superset of its correct atomic translators Λ_k. The intuition behind candidate generation is, despite the complexities of IR libraries, we observe that a correct atomic translator should first be well-typed. For instance, in Fig. 4, the atomic translator for branch instruction should initially take the parameter with type `Branch_s` and through the utilization of APIs, finally produce the result with type `Branch_t`. To this end, we employ the well-studied component-based synthesis [25, 46] to analyze the type information of APIs and generate the initial candidates for us.

Specifically, to generate such well-typed candidates atomic translators, SIRO first encodes the required types and APIs onto a graph in Def. 4.1, where the edges represent the consumption and production relations between APIs and types.

Definition 4.1 (IR Type Graph)**.** Given the IR libraries and operand translators provided by our skeleton, the IR type graph is defined as a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. Specifically:

- Node Set: $\mathcal{N} = \mathcal{A} \cup \Omega$, where $a \in \mathcal{A}$ represents an API function (IR getters, IR builders, and operand translators) and $\omega \in \Omega$ represents a type.
- Edge Set: $\mathcal{E} \subset \mathcal{N} \times \mathcal{N}$. A return edge $a \rightarrow \omega \in (\mathcal{A} \times \Omega)$ indicates that an API function a produces a type ω as its return value. A parameter edge $\omega \xrightarrow{x} a \in (\Omega \times \mathcal{A})$ indicates that a type ω can be consumed by an API function a as a parameter. The edge label $x \in \mathbb{Z}^+$ differentiates the parameter positions.

Subsequently, the candidates can be generated by searching for the so-called feasible subgraphs in Def. 4.2. Essentially, a feasible subgraph represents a valid implementation of atomic translator that conforms to the type signature using APIs, and can be searched out via backward BFS from the target type. In the supplementary material, we have provided an example to illustrate the feasible subgraphs related to Fig. 4.

Definition 4.2 (Feasible Subgraph)**.** Consider an IR type graph $\mathcal{G}$ and a common instruction kind k, where the kind k has types $\omega_k, \omega'_k \in \Omega$ defined in source and target IR libraries. A subgraph $\mathcal{G}' = (\mathcal{A}' \cup \Omega', \mathcal{E}') \subset \mathcal{G}$ is a feasible subgraph w.r.t $\mathcal{G}$ and k, iff it obeys two rules:

- **Consumption Rule.** For every API $a' \in \mathcal{A}'$, it has precisely n incoming edges in $\mathcal{G}'$ where n equals the parameter number of a'. In addition, the labels of these incoming edges span from 1 to n. This rule ensures that each invocation of an API consumes the appropriate types.

Algorithm 3: Form of a per-test translator

```
1 Function TranslateInst(i):
2     if loc(i) = 1 then i' ← λ1(i);
3     else if loc(i) = 2 then i' ← λ2(i);
4     ...;
5     else if loc(i) = m then i' ← λm(i);
6     return i';
```

- **Reachability Rule.** Firstly, the source and target types ω_k, ω'_k belong to Ω'. Secondly, for each type $\omega' \in \Omega' - \{\omega'_k\}$, there exists a reachable path p in $\mathcal{G}'$ from ω' to ω'_k. It ensures that the target type ω'_k is obtained by consuming the other types in $\mathcal{G}'$ including ω_k.

4.3 Test-guided Synthesis

Regarding the candidates Λ^* generated in § 4.2, two issues remain unresolved: (a) identifying the correct candidates. (b) associating a correct candidate with its corresponding predicate. To answer them, SIRO profiles each test case (§ 4.3.1) to enumeration per-test case translators (§ 4.3.2), validates them (§ 4.3.3) to refine the candidates (§ 4.3.4), and outputs complete instruction translators (§ 4.3.5).

4.3.1 Profiling. Profilers provide essential information to construct per-test translators for each test case by scanning involved instructions, and can be automatically generated given the IR library. To obtain different information, we designed three profilers shown as follows:

- ***Location Profiler*** is used to label each involved instruction with a unique identifier. It can be achieved by tagging the instruction based on its traversal order.
- ***Kind Profiler*** is used to determine the kind of each involved instruction. Its implementation is similar to lines 18-20 in Alg. 1, where the instruction kind is distinguished by consecutive type checks.
- ***Sub-kind Profiler*** obtains the runtime values of the predicates of each instruction. It is achieved by invoking the bool/enum IR getters. For instance, for a specific unconditional branch instruction in Fig. 4, the sub-kind profiler can obtain `IsUncondBr() == True` for it.

We organize the profiling data into a profile table (Def. 4.3) to guide further per-test translator enumeration.

Definition 4.3 (Profile table)**.** For a test case t with m instructions, the profiling results are defined as a table $\tau_t : \mathbb{Z}^+ \mapsto (K \times \Sigma^{\&})$. Each instruction in t is denoted as $l \rightarrow (k, \sigma^{\&})$, where $l \in \mathbb{Z}^+$ denotes the unique location, $k \in K$ identifies its kind, and $\sigma^{\&} \in \Sigma^{\&}$ records the conjunction of all possible predicates at runtime.

4.3.2 Enumeration. The enumeration utilizes the profile table to compose a set of "ad-hoc" per-test translators PT_t. We slightly modify `TranslateInst` in Alg. 1 to Alg. 3 to form a per-test translator. Notably, for each location $x \in \{1 \ldots m\}$,

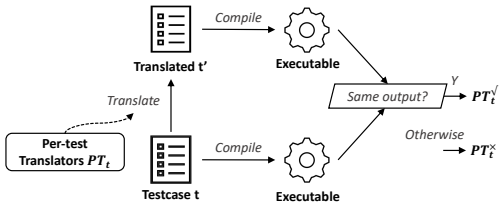

Figure 6. Validate per-test translators via differential testing

there is a box $\boxed{\lambda_x}$, meaning that it could be filled with possible candidate atomic translators. By enumerating these boxes in each location, per-test translators can be obtained as a set of lists (Def. 4.4). For example, if a test case t includes three instructions, which have 3, 4, 5 candidates according to their kinds, then $60 (= 3 * 4 * 5)$ per-test translators would be generated for validation.

Definition 4.4 (Per-test Translators)**.** Given a test case t with m instructions, the per-test translators after enumeration is a set PT_t. Each element $pt \in PT_t$ is a list $\{\lambda_1 \dots \lambda_x \dots \lambda_m\}$, where λ_x is an atomic translator candidate assigned for instruction at location x, enumerated from set $\Lambda^*_{\tau_t[x].k}$.

4.3.3 Validation. To validate per-test translators, we adapt differential compiler testing [50, 53, 90] to the context of IR translation. As shown in Fig. 6, a per-test translator $pt \in PT_t$ would be placed into a set $PT_t^{\surd}$ if it successfully finishes the translation and compilation, and passes the execution oracle. Specifically, the test cases are designed in the form of an IR program of source version that includes a main function. The main function returns a constant value during execution with no inputs, which is given along with the test case as the oracle. Such a form is commonly found in the test suites of mainstream compilers. Overall, the validation process provides us with a criterion: if a candidate can correctly translate an instruction i in test case t, then it must have participated in at least one of $PT_t^{\surd}$.

Indeed, it is possible to replace this validation approach with translation validation [65, 66], which could hopefully reduce the average size of $PT_t^{\surd}$. Unfortunately, the translation validator itself could fall into the IR version trap, e.g., the cutting-edge tool Alive2 [65] fails to support multiple IR versions. Moreover, the constraint solving makes the validation process time-consuming. Hence, for practicality and efficiency, we chose the current testing-based approach.

4.3.4 Refinement. The refinement in Alg. 4 uses a mapping $\mathbb{M}^*_k : [\Sigma^{\&}_k \mapsto \Lambda^*_k]$ for each kind k, conservatively recording the correct candidates and their predicates. For each instruction at location x in the test case t, we collect the refined candidates $\Lambda^{\surd}_x$ that participate in the successful translation $PT_t^{\surd}$ and obtain the kind k and conjuncted predicates $\sigma^{\&}$ of the instruction by querying τ_t (line 3). Then, $\mathbb{M}^*_k$ is updated with additional predicates and refined candidates (lines 4-5).

Algorithm 4: Refining the atomic translators

```
1 Function Refine(M*, PT_t^✓, τ_t):
2     for x ∈ 1...m do
3         Λ_x^✓ ← {pt[x] | pt ∈ PT_t^✓}; (k, σ^&) ← τ_t[x];
4         if σ^& ∉ keys(M*_k) then M*_k[σ^&] ← Λ_x^✓ ;
5         else M*_k[σ^&] ← M*_k[σ^&] ∩ Λ_x^✓;
```

```
1 define i32 @main() {
2   entry:
3     ; ...
4     ; %a == 10, %b ==10
5     %ret ← %a - %b
6     return %ret
7 } ; oracle: 0
```

```
1 define i32 @main() {
2   entry:
3     ; ...
4     ; %c == 20, %d ==10
5     %ret ← %c - %d
6     return %ret
7 } ; oracle: 10
```

Figure 7. Two simplified test cases. Before line 5, in the left case, variables a and b are both evaluated to 10, resulting in a return value of 0. In the right case, variables c and d are evaluated to 20 and 10 respectively before line 5, leading to a return value of 10.

Note that in addition to those entirely correct per-test translators, the set $PT_t^{\surd}$ may contain per-test translators that are composed of some incorrect atomic translators. Consider the subtraction instruction %ret ← %a - %b in the left test case of Fig. 7. If an incorrect atomic translator translates it to %ret ← %a - %a with other instructions being correctly translated, the return value of the translated test case remains 0. In this case, this atomic translator will not be refused because our refinement process only rejects atomic translators that are definitely incorrect, i.e., not participating in any successful translation. Hence, to refuse such incorrect atomic translator, one needs to provide a test case such as the right case in Fig. 7. That's because when line 5 is incorrectly translated to %ret ← %c - %c, the latter would be evaluated to 0 during execution, which violates the oracle.

4.3.5 Skeleton Completion. Once all the test cases have been processed, we generate for each instruction kind k the instruction translator $\mathbb{M}_k$ from $\mathbb{M}^*_k$. In fact, the correct atomic translators for kind k are recorded in the mapping values of $\mathbb{M}^*_k$, each of which is a set Λ. Specifically, to generate $\mathbb{M}^*_k$, we first iterate through each Λ and check if there exists an atomic translator λ that belongs to each Λ, indicating the kind k has only one sub-kind and $\mathbb{M}_k$ is assigned as $[true \rightarrow \lambda]$. Otherwise, we select the minimum number of atomic translators that can cover all the mapping keys, i.e., all the encountered conjunctive predicates. For each selected atomic translator, we perform a logical OR operation on the conjunctive predicates it covers to remove the irrelevant predicates and obtain the most accurate one.

For unseen conjunctive predicates, we perform a logical OR operation on them and generate an if statement that

triggers a warning. When using our tool on an actual program, if we encounter a new conjunctive predicate that is not covered by any test case, it helps us identify the exact location of the error quickly and prompts users to add a new test case to address the issue.

4.4 Optimizing Strategies

A significant portion of the synthesis process is dominated by validating per-test translators. However, we have observed that many of these validations are redundant. To enhance the performance of Siro, we have implemented three optimizations to eliminate these redundancies.

Optimization I (Equivalence). We utilize the equivalences found in the profile table to merge per-test translators that have the same effect. First, if two instructions have the same conjunctive predicate at runtime, indicating the same subkind, we can always assign them with the same atomic translator during enumeration. Second, we noticed that some IR getters are aliases of others. We can extend the profile table to record the objects returned by IR getters and merge equivalent translators based on the equivalence of these objects.

Optimization II (Memoization). Since $\mathbb{M}^*$ remembers the encountered conjunctive predicates and the corresponding refined candidates, we can utilize it during the enumeration process to perform pruning. Specifically, when encountering an instruction of kind k in the test case, if its conjunctive predicate $\sigma^{\&}$ exists as a key in $\mathbb{M}^*_k$, we can efficiently perform the enumeration by directly applying $\mathbb{M}^*_k[\sigma^{\&}]$.

Optimization III (Test Case Order). Building upon optimization II, we discovered that placing simpler test cases earlier helps refine candidates efficiently, allowing subsequent complex test cases to leverage optimization II with $\mathbb{M}^*$. To implement this optimization, we utilize a lightweight heuristic by establishing a topological order based on the kinds of instructions present in each test case.

5 Implementation

We implemented Siro on LLVM infrastructure and its IR, which numerous IR-based software rely on [14, 52, 75, 80]. We leveraged LLVM's C++ IR libraries as the materials of IR translation, while the synthesizer for instruction translators was implemented in Python. Next, we discuss several challenges we solved during implementation.

Programming with Two Versions. Notice that the materials of the IR translator include different versions of IR libraries. In order to program with both versions simultaneously, we refactored the namespace of one version's IR library through simple string replacement.

Mapping between IR Instructions. To determine the corresponding source and target instruction types between versions, Siro automatically matches the type signature of instructions utilizing the IR libraries of two versions. In Fig. 4, we use two symbols `Branch_s` and `Branch_t` to represent the same instruction in different versions, but in actual implementation, they share exactly the same type of signature. Additionally, Siro would automatically report a new instruction to developers to handle when there is no match for an instruction type.

Handling IR Value Dependence. Our translation algorithm for IR is a one-pass traversal, where we need to handle the dependencies between IR values, e.g., an operand of an instruction may be another instruction or a function that has not been translated yet. To handle this issue, we maintain a mapping to record the correspondence between the values of two IR versions. If we encounter an operand that has not been translated, a placeholder value in the target version is generated. Once the translation for that operand is completed, we replace all usages of the placeholder with the actual translated result.

Speeding up Synthesis Process. We realized that sequentially generating code and compiling executables for each per-test translator would lead to lengthy build times. Therefore, we have designed a wrapper function that accepts a list of runtime parameters to specify the per-test translators to be validated. This allows us to compile atomic translators only once and dynamically enumerate per-test translators, significantly increasing the efficiency. Further, we parallelized the translator validation, observing no interdependencies between them. This dramatically speeds up Siro.

6 Evaluation

We evaluated Siro by investigating three research questions.

- **RQ1.** Can Siro synthesize IR translators between different IR versions?
- **RQ2.** How do the IR translators produced by Siro benefit different program analysis tasks?
- **RQ3.** How does each counterpart of Siro benefit the overall performance of synthesis?

We conducted all experiments on a 64-bit machine with a 20-core Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz and 256G of physical memory. When parallelizing the validation process, the thread number was configured to be 40.

6.1 LLVM Version Upgrades

To illustrate the evolution of LLVM IR and guide our version pair selection, we studied the version changes of LLVM IR from 3.0 to 17.0, spanning a period of 12 years (2011 – 2023). Specifically, we studied the three sources of incompatibilities presented in § 3.1 by examining the release notes and GitHub repository of different versions of LLVM IR. For text incompatibility, we evaluated the changes in the implementation codes of LLVM's bitcode parser and reader. For API incompatibility, we monitored the changes in the C++ headers related to LLVM IR, and importantly, we counted changes in the implement codes of representative LLVM's built-in static analyses (alias, dependence, and dominance analyses).

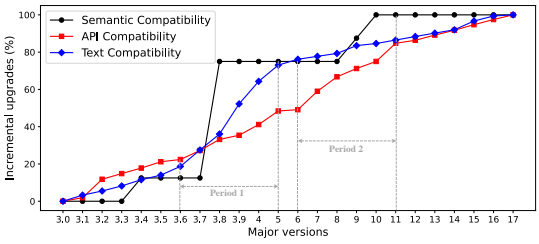

Figure 8. The overall upgrading trend of LLVM IR

For semantic incompatibility, we recorded the appearance of new instructions in different versions.

Results. The study results are presented as follows. Overall, the text and API dimensions involved approximately 25 KLOC and 31 KLOC code changes, respectively. Meanwhile, the semantic dimension witnessed the birth of 8 new instructions. Additionally, we use a cumulative line graph in Fig. 8 to provide a more straightforward visualization of LLVM IR's upgrading trends. The X-axis represents the major versions we analyzed[1], while the Y-axis represents the incremental changes of a specific dimension throughout the overall evolution. More precisely, the percentage increase on the Y-axis for each version is calculated by dividing the number of updates (lines or instructions) for a single version by the total number of updates for all versions, i.e., the contribution of that version to the overall version changes. To normalize the differences between various modules within a dimension, such as the bitcode parser and reader, we calculated the percentage for each module, assigned them equal weights, and then obtained the overall percentage. It can be observed that, during the continuous evolution of LLVM IR, there are two periods with relatively active growth. The first period, from 3.6 to 5, witnessed significant updates across all three dimensions. The second period, from 6 to 11, involved substantial updates in the API and semantic dimensions.

6.2 RQ1: Effectiveness

To demonstrate the effectiveness of Siro as a synthesis-powered framework, we attempt to develop IR translators of diverse source-target version pairs. As shown in Tab. 3, we have selected ten different source-target version pairs as goals based on the study results. These pairs encompass various scenarios. Pairs 1-6 represent a "long distance" of version gaps, including the two growth periods. Pairs 7-9 concern the relatively close versions, and the last pair handles the translation from the low version to the high version.

Involved Manual Efforts. By utilizing Siro, all ten IR translators were efficiently produced. This part specifically elucidates how SIRO alleviated the burden of manual development. During the experiment, we engaged a Ph.D. student with a background in C++ programming but no prior knowledge of Siro's internal design as a user of the system. Recall that in Siro, a user needs to provide test cases to synthesize common instruction translators and handle a few new instructions (§ 4.1). Overall, these tasks were completed by the student within a span of ten days, with two hours dedicated each day. For each version pair, Siro completed the synthesis of common instruction translators within a timeframe of less than three hours, based on the established test cases.

[1]Note that before LLVM 4, the decimal point of the version number denotes the major version, such as 3.x

Table 3. Pairs of IR translator versions achieved by Siro

No.	Source Version	Target Version	# Common Inst	# New Inst	#Atomic Trans (LOC)	#Inst Trans (LOC)
1	12.0	3.6	58	7	71,900	783
2	13.0	3.6	58	7	74,240	783
3	14.0	3.6	58	7	73,100	780
4	15.0	3.6	58	7	80,293	781
5	17.0	3.6	58	7	80,293	781
6	17.0	3.0	57	8	76,248	768
7	3.6	3.0	57	1	76,212	768
8	5.0	4.0	63	0	72,086	843
9	17.0	12.0	65	0	84,730	875
10	3.6	12.0	58	0	73,460	785

Altogether, a total of 68 test cases were employed with Siro to achieve the correct synthesis of all the common instructions across all version pairs. These test cases were primarily derived from two sources. Firstly, for simple instructions like arithmetic operations, small C test cases that can be compiled into IR programs were written. Secondly, for more intricate instructions, the official LLVM test suite [61] proved to be a valuable resource. By searching over the test suite, multiple test cases could be extracted at the level of IR programs. In particular, 60 test cases were initially provided for the first version pair, and these test cases were subsequently reused with minor textual modifications for the remaining version pairs. Only for the eighth and ninth pairs did the student introduce eight additional test cases to cover the seven common instructions. Moreover, Siro played a crucial role in assisting the student in identifying and removing duplicated test cases by reporting whether specific test cases have effectively pruned out some candidates.

Furthermore, through the interaction with Siro, the student was able to contribute high-quality test cases. An illustrative example is presented in Fig. 10. The user initially writes a test case with execution oracle 42, i.e., the code containing line 4 before the difference. However, this particular case fails to refuse the two erroneous atomic translators depicted in Fig. 9. Compared with the correct implementation (lines 7-13 in Fig. 4), AtomicBranch1 incorrectly translates the second branch target, while AtomicBranch2 wrongly interchanges the two branch targets. For AtomicBranch1, The underlying reason is that the provided test case only exercises the true branch. As for AtomicBranch2, it can be combined with another atomic translator of compare instruction (icmp) which incorrectly swaps the operands, to pass the provided test case. Upon observing the contradictions among the remaining atomic translators, the user can refuse the incorrect translators by providing an additional test case. For instance, one approach could involve replacing line 4 of

```
1  Branch_t AtomicBranch1(Branch_s inst) {
2      Value_s cond = inst.GetCond();
3      Block_s b0 = inst.GetBlock(0);
4      Value_t condd = TranslateValue(cond);
5      Block_t bb0 = TranslateBlock(b0);
6      return Builder.CreateCondBr(condd, bb0, bb0);
7    }
8  Branch_t AtomicBranch2(Branch_s inst) {
9      Value_s cond = inst.GetCond();
10     Block_s b0 = inst.GetBlock(0);
11     Block_s b1 = inst.GetBlock(1);
12     Value_t condd = TranslateValue(cond);
13     Block_t bb0 = TranslateBlock(b0);
14     Block_t bb1 = TranslateBlock(b1);
15     return Builder.CreateCondBr(condd, bb1, bb0);
16   }
```

Figure 9. Two incorrect atomic translators for conditional branch instruction. The incorrect parts are highlighted in yellow.

```
1  define i32 @main() {
2    entry:
3      ;...
4  -   %cond = icmp eq i32 10, i32 10
5  +   %cond = icmp eq i32 10, i32 20
6      br i1 %cond, %then, %else
7    then:
8      ret i32 42
9    else:
10     ret i32 41
11 }
```

Figure 10. An example of two test cases. The initial case (before diff) can't refuse the two incorrect candidates in Fig 9, which can be pruned out by the enhanced test case (after diff).

the initial test case with line 5 and accordingly updating the execution oracle to 41.

The correctness of the instruction translators generated by Siro was confirmed through our manual review process. The last two columns of Tab. 3 present for each version pair, the code size of the initially generated atomic translators (column #Atomic Trans) and the final instruction translators (column #Inst Trans) in the synthesis procedure. The review process for each instruction translator focused on three aspects. Firstly, we ensured the correct identification of predicates. Secondly, we examined the correctness of the atomic translators under each predicate. Lastly, if multiple atomic translators existed under a predicate, we assessed their equivalence of implementation. For example, during the review of the translator depicted in Fig. 11, we first confirmed the correctness of the predicates and subsequently verified the correctness of the two atomic translators in lines 3-5 and 7-13. Finally, we determined that these two atomic translators were equivalent to those depicted in Figure 4,

```
1  Branch_t TranslateBranch2(Branch_s inst) {
2    if(inst.IsUncondBr()) {
3      Block_s b = inst.GetOperand(0);
4      Block_t bb = TranslateBlock(b);
5      return Builder.CreateBr(bb);
6    } else {
7      Value_s cond = inst.GetCond();
8      Block_s b0 = inst.GetOperand(1);
9      Block_s b1 = inst.GetOperand(2);
10     Value_t condd = TranslateValue(cond);
11     Block_t bb0 = TranslateBlock(b0);
12     Block_t bb1 = TranslateBlock(b1);
13     return Builder.CreateCondBr(condd, bb0, bb1);
14   } }
```

Figure 11. Another correct instruction translator for the branch instruction. Compared with the one in Fig 4, it is implemented using equivalent getter APIs highlighted in yellow. The type conversion operations are omitted for brevity.

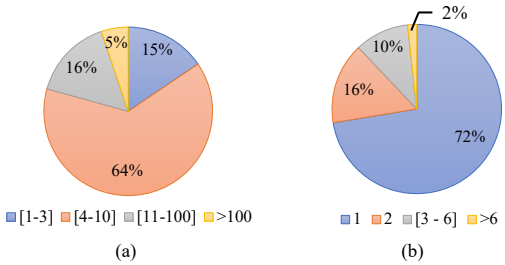

Figure 12. The number distributions of the candidate and refined atomic translators

where the API GetOperand() was found to be equivalent to GetBlock() for retrieving the desired branch targets.

Synthesizing Common Instructions. We illustrate the effectiveness of Siro using statistics regarding the atomic translators. Fig. 12 displays the number distribution of candidates and refined atomic translators for all common instructions between versions 12.0 and 3.6, i.e., the initial search space and the final solution space. As shown in Fig. 12(a), the search space remains far from trivial, even when using type-guided synthesis to generate candidates. Only 15% of the instructions have three or fewer candidates, while the majority of instructions have four to ten candidates. Additionally, a significant portion (21%) of instructions have dozens or even hundreds of candidates. Moreover, considering the combination of instructions further amplifies the complexity of the search space, leading to an even larger space to explore.

Fig. 12(b) shows that after the synthesis, most instructions (72%) remain one atomic translator, indicating the resulting instruction translator has no predicates. For those instructions with more than one refined atomic translator, some

```
Invoke_t TranslateInvoke(Invoke_s inst) {
  Function_s fun = inst.GetFunc();
  Function_t func = TranslateFunction(fun);
+ Type_s ty = fun.GetType();
+ Type_t type = TranslateType(ty);
  /*...*/
- return Builder.CreateInvoke(func, ...);
+ return Builder.CreateInvoke(func, type, ...);
}
```

Figure 13. Variations of invoke instructor translator due to API changes.

interesting findings are discovered: First, SIRO identifies subkinds for several instructions, such as `branch`, `return`, and `call`. Their instruction translators utilize predicates to distinguish different atomic translators. Second, SIRO has "found" the commutative properties of some instructions. For example, when handling arithmetic instructions like `add` and `mul`, swapping their operands during translation results in an equivalent instruction. Third, there are cases where an instruction has multiple builders, and more than one of them can be used to correctly construct the instruction. These intriguing results are all automatically synthesized by SIRO using test cases, relieving users from the need to delve into these implementation details.

Additionally, for the same instruction translator generated by SIRO, we have observed that its implementation differs across different versions. For instance, since LLVM 9.0, the builder of invoke instruction requires explicitly specifying the type of the called function. Therefore, in version pairs whose targets >= 9 (i.e., pairs 9-10), the translators resemble the code before the difference in Fig. 13, otherwise they resemble the code after the difference. This reflects how program synthesis techniques have helped SIRO effectively overcome API incompatibility in the IR libraries.

Handling New Instructions. We have dealt with eight new instructions in pairs 1-7 using the principles in § 3.3.2. Specifically, we found that five instructions are unnecessary to process because they are related to error mechanisms in Windows platform [64] and are never encountered in Linux. Then, we applied analysis-preserving translations to the remaining three instructions. For the `callbr` instruction used to model possible jump targets in inline assembly, we translated it into normal inline assembly along with a switch statement to restore its control flow. For the `freeze` instruction used to prevent the propagation of a possible undefined behavior value [51], we translated it into its operand value to preserve the data flow. For the `addrspacecast` instruction introduced in LLVM 3.4, we translated it back into the `bitcast` instruction in pair 7, which was the original way of using it before LLVM 3.4. Pairs 8-10 don't need any special treatment for new instructions because either the two versions are close or the higher version already includes all the instructions of the lower version.

6.3 RQ2: Benefit to Program Analysis

To quantify the benefits and prospects of IR translation technique for IR-based software, we conducted comprehensive experiments on three clients including static bug detection, fuzzing, and kernel bug detection utilizing the IR translators produced by SIRO.

Static Bug Detection. We evaluated the value flow analysis tool PINPOINT [82] developed on LLVM 3.6, under two settings: one uses the compiling approach, i.e., Clang 3.6, to obtain IR programs, the other uses the translating approach, utilizing Clang 12.0 compiler and 12.0 → 3.6 translator produced by SIRO. When running PINPOINT on several fundamental open-source projects, We choose four common kinds of bugs, namely, null pointer dereference (NPD), use-after-free (UAF), file descriptor leak (FDL), and memory leak (ML). To determine whether a bug is reported under both settings, we compare each step in the bug report trace by the file name, line number, and descriptions.

Tab. 4 shows the comparison results of reported bugs under two settings. Specifically, the columns **new**, **miss**, and **shared** represent the number of bugs reported only by translating approach, only by compiling approach, and by both approaches. As shown in the table, 91% (=253/276) bugs overlap when analyzing the two forms of LLVM 3.6 IR programs. Meanwhile, the translating and compiling approaches independently identified 15 and 8 bugs, respectively. This primarily stems from the fact that IR programs generated from different compiler versions are distinct, enabling static analyzers to uncover different bugs. Similar findings have been reported by several recent studies [55, 76]. These promising results provide strong empirical evidence that the IR translation technique can help bug detectors overcome the version trap and accurately identify bugs.

Fuzzing. To quantify the benefit to fuzzers, we choose a recently popular fuzzing benchmark MAGMA [40]. Our goal is to trigger all 111 CVEs in the benchmark programs using the executables compiled from the IR programs translated by our 12.0 → 3.6 translator. Particularly, MAGMA is built upon real-world open-source projects and includes the inputs, known as PoCs, that can trigger the crashes related to each specific CVE. A single CVE can correspond to multiple PoCs.

As shown in Tab. 5, 95.89% (35299/33849) PoCs can be reproduced, which triggers 85.39% (95/111) of all the CVEs in the benchmark projects. For the test cases underlying the project php, the IR translator succeeded in translating it, but the compiler crashed during the backend code generation phase. The cause is that php hard-codes hardware instructions in source code using inline assembly, and is only supported by higher-version compiler backends. This goes beyond the scope of what IR translation can handle. Overall, the high success ratio of reproducing PoCs suggests that IR

Table 4. Bugs reported by Pinpoint under two settings

Project	NPD			UAF			FDL			ML		
	new	miss	shared	new	miss	shared	new	miss	shared	new	miss	shared
libcapstone	1	0	18	0	0	0	0	0	0	0	0	0
tmux	2	0	85	0	3	14	0	0	0	9	5	105
libssh	3	0	21	0	0	0	0	0	0	0	0	4
libuv	0	0	0	0	0	2	0	0	0	0	0	0
pbzip	0	0	0	0	0	0	0	0	0	0	0	0
libcjson	0	0	0	0	0	0	0	0	0	0	0	0
http-parser	0	0	0	0	0	0	0	0	0	0	0	0
pkg-config	0	0	3	0	0	0	0	0	1	0	0	0
Total	**6**	**0**	**127**	**0**	**3**	**16**	**0**	**0**	**1**	**9**	**5**	**109**

Table 5. Statistics of reproducing PoCs with Siro

Project	#Targets	#Insts	#CVE	#PoC	#R-CVE	#R-PoC	CVE-Ratio	PoC-Ratio
libpng	1	109,123	7	634	7	634	100.00%	100.00%
libtiff	2	557,882	14	3,716	14	3,709	100.00%	99.81%
libxml	2	2,076,676	15	19,731	15	19,731	100.00%	100.00%
poppler	3	3,786,354	19	7,343	19	7343	100.00%	100.00%
openssl	4	5,508,226	20	655	20	655	100.00%	100.00%
sqlite	1	692,561	20	1,777	20	1,777	100.00%	100.00%
php	1	2,470,023	16	1,443	0	0	0.00%	0.00%
Total	-	-	**111**	**35,299**	**95**	**33,849**	**85.59%**	**95.89%**

translation holds great promise for dynamic analysis such as grey-box fuzzing, where static analysis on different versions can collaborate to guide the fuzzing process.

Handling Linux Kernel. Finally, we applied IR translation in the context of bug detection for the Linux kernel in real-world scenarios, where the complexity of the Linux kernel makes it impossible for the compiling approach to obtain a complete IR program. Fortunately, through the utilization of our 14.0 ⟶ 3.6 and 15.0 ⟶ 3.6 translators, we successfully obtained the corresponding IR programs. This allowed us to leverage existing value flow analysis techniques [82] and function pointer analysis [67] to develop a similarity-based kernel bug detector. Specifically, this tool utilizes existing security patches in Linux drivers to reason about the root causes of the bugs and detect potential similar bugs in other drivers via value flow path searching. As a result, we were able to identify and confirm the presence of 80 previously unknown bugs. Moreover, all of them have been confirmed by kernel developers and 56 of them have been fixed and merged by our patches. This notable achievement showcases how IR translation has practically aided researchers in tackling the IR version trap.

6.4 RQ3: Performance

To demonstrate how our technical design improves the synthesis performance of Siro, we analyzed the overall time breakdown and conducted an ablation study on a version pair (13.0 ⟶ 3.6) utilizing the 60 test cases we gathered.

Time Breakdown. Overall, it took 2.91 hours to complete the synthesis. During this time, 90.7% was spent on the validation of per-test translators, while the remaining time was divided into 0.12 hours for enumeration and 0.15 hours for refinement and skeleton completion. This is thanks to implementing the per-test translators as a wrapper function, which significantly reduces the compilation time during enumeration, enabling us to allocate more computational resources to validation. In particular, within the validation process, only 0.19 hours were spent on running test cases to check the execution results. This reveals that the success or failure of translation and compilation also plays a crucial role in ensuring correctness, helping us reject a significant number of erroneous per-test translators at an early stage.

Ablation Study. We conducted three experiments to shed more light on the operation of Siro. First, we attempted to not use per-test translators by enumerating all possible translators for each instruction together. However, even without considering predicates, the number of combinations reached 10^{40}, leaving no chances for synthesis. Second, we disabled optimizations I and II discussed in § 4.4. Unfortunately, the synthesis process got timeout after 24 hours, stuck on a test case with 13,000,000 per-test translators waiting to be validated. This reflects the importance of our designed optimization, especially in applying the memorized knowledge in $\mathbb{M}^*$ immediately after refinement. Third, we compared optimization III in § 4.4 with five randomly generated order sequences. Consequently, three sequences got timeout after 24 hours and two sequences succeeded in 5.06 hours and 8.23 hours, proving that our order strategy can effectively reduce redundant validations.

7 Discussion

Limitations of Proposed Approach. Siro utilizes program synthesis to automatically generate translators for common instructions in different versions. The underlying assumption is that there exists a significant amount of one-to-one mapping in the version trap scenario. However, if the synthesis goal is to find translators between IRs (e.g., for different languages) [5, 29], Siro cannot be directly adopted. In such cases, the form of translation needs to be extended to support one-to-many or many-to-many mappings. Nonetheless, the insights in the algorithmic skeleton and the per-test translator we proposed are still applicable.

Siro reduces most of the laborious work for developers by extracting the version-agnostic skeleton and synthesizing common instructions. However, developers still need to collect test cases for common instructions and write translations for new instructions. In the pursuit of fully automating IR translations, we have identified two challenging directions for future researchers. First, existing test program generation techniques [11, 57, 90] face difficulties in achieving diversity in IR instructions. Second, it is worth exploring the extraction of IR semantics (via techniques such as symbolic execution [9] and translation validation [65]) to assist in automatically handling translations for new instructions.

Generalizability of Siro. While Siro is instantiated on LLVM IR, our proposed method can be generalized to enhance the version compatibility of other IRs as well. When

handling a new IR, one can start by identifying the five types of IR libraries mentioned in Tab. 2 within this IR ecosystem. Subsequently, the corresponding version-agnostic translation skeleton can be achieved based on a specific IR structure, e.g., the control flow graph. In addition, following the approach outlined in § 4, the synthesis system can be established. From the perspective of data structures, SIRO first divides and conquers complex structures, leaving structurally simple but content-rich data to program synthesis. This principle is also applicable to the synthesis of other data structures, such as network protocols and serialization data formats.

Suggestions to Developers. To avoid falling into the IR version trap, developers of IR-based software can consider the following approaches. First, reducing the coupling between program analysis and the underlying IRs would be advantageous. This can be achieved by either introducing a generic interface to abstract the IR [20, 21, 85], or decomposing a program analysis algorithm into generic interfaces for specific IRs to achieve [39, 83]. Second, it is crucial to have a regression test suite [38, 77] for a program analysis tool, as all three approaches, i.e., upgrading, compiling, and translating, we discussed can leverage this test suite effectively.

A positive example is the efforts of the OCaml community in ppxlib [70]. This library strives to maintain a version-independent IR and unifies different versions of the OCaml front-end IR onto that IR. Additionally, it provides a set of version-stable APIs for developers to implement their static analyzers and rewriters. With these two layers of abstraction, the main burden of version maintenance falls on the developers of ppxlib, while the developers of downstream applications can enjoy great version compatibility. Furthermore, the subsequent extension of LLVM, MLIR [24, 49], holds similar potential. MLIR introduces the concept of dialects, allowing the presence of various abstraction levels within the IR, which then allows static analysis to manipulate the underlying IR in a generic programming fashion and focus on specific characteristics.

8 Related Work

IR and IR-based Software. Bridging the source code and binary executables, Intermediate Representation (IR) is a mid-level program abstraction produced by compilers [15]. The emergence of IRs has spawned plenty of IR-based software to analyze [9, 23] and transform [1, 27] programs. To ease the manipulation of code, researchers have designed various IR structures that highlight different program relationships. The most typical ones include abstract syntax tree [6] for syntactic scoping, control flow graphs [19, 32] for execution order, static single assignment [7, 18, 84] for variable def-use relation, and dependence graphs [16, 26, 69, 82] for data and control dependence.

Considering the evolution of IRs over versions hurts the usability of IR-based software, Siro is proposed to resolve the version compatibility issues. In Siro, the translation algorithm related to IR structures is settled into our skeleton (§ 3.2), allowing us to concentrate on handling the diverse IR instructions. This method ensures the generality across various IR structures.

Program Transformation. Program transformation modifies a given program to enhance its portability, performance, and readability, which includes program migration [81], version upgrade [2, 79], and even program optimization [47, 86]. According to technical designs, previous studies can be divided into three categories. The first line leverages predefined rules to achieve the transformations, such as converting imperative code to functional code [37], and translating concurrent C code to rust code for safety [41]. The second category takes the program facts obtained by static [87] or dynamic analysis [68, 81] as the semantic restriction and synthesizes a new program preserving the semantic equivalence. The third body of the literature adopts the NLP techniques, such as translating Python between versions [2], and transforming the instructions targeting different architectures [88].

Specifically, Siro is a framework for transforming between different IR versions. It encompasses the first two techniques, where we employ one-to-many rules to translate new instructions and we leverage the dynamic execution results to effectively synthesize common instruction translators. Additionally, it would also be promising to adopt new advances, such as the large language model [8, 71] to resolve the version upgrade problem, which could enhance Siro to achieve semantic-equivalent translation.

Component-based Program Synthesis. From a technical perspective, our work is an instance of component-based program synthesis [33, 43]. Generally, the synthesizers compose several components, such as classes and methods in the libraries, to achieve desired functionalities, which are mostly specified by input-output examples [25, 46]. To prune search space, existing techniques often leverage the type signatures to narrow down the possible combinations of library function calls [35, 36, 72]. For example, SyPet [25] encodes the type signature of each library function with a Petri net, and attempts to enumerate well-typed Java programs passing given test cases based on the reachable paths of the Petri net. Other efforts such as APIphany [34] and Hoogle+ [43] follow a similar spirit to SyPet, concentrating the component-based synthesis in different real-world scenarios.

In Siro, we utilize the type signatures of IR builder and getter APIs to generate candidate atomic translators (§ 4.2), sharing the spirit of component-based synthesis. However, it is specifically our decomposition of the translation algorithm that enables atomic translators to be viable clients of component-based synthesis. Moreover, our contribution to synthesis is also reflected in the novel validation approach through the per-test translators (§ 4.3). In previous works,

each test case is used to validate one task individually. But in Siro, the correctness of each instruction translator cannot be validated independently and each test case is used to validate multiple instruction translators as well. Lastly, the three optimization strategies we propose greatly improve the performance (§ 4.4). Without the collaboration between these technical designs, the program synthesis technique cannot be practically adopted in the scenarios of IR translation.

9 Conclusion

We have presented Siro, the first synthesis-powered framework to efficiently generate IR translators, thereby enhancing IR version compatibility. Siro employs a divide-and-conquer approach to establish an algorithmic skeleton and leverages type-guided generation and test case-guided refinement to automatically synthesize IR instruction translators. Our experiments show that Siro can generate multiple well-functional IR translators, effectively supporting static analyzers and fuzzers to find bugs in real-world programs such as the Linux kernel. We believe that Siro can make program analysis techniques more practical by assisting the developers in escaping the IR version trap.

Acknowledgments

We would like to acknowledge and thank the anonymous reviewers and our paper shepherd, Prof. Lulian Neamtiu, for providing constructive suggestions and insightful feedback. This work is suppported by ITS/440/18FP grant from the Hong Kong Innovation and Technology Commission, National Natural Science Foundation of China under Grant No. 62302434, and research grants from Huawei, Microsoft, and TCL. Wei Chen is the corresponding author.

References

[1] AFL. American fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2022.

[2] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. Using machine translation for converting python 2 to python 3 code. *PeerJ Prepr.*, 3:e1459, 2015.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, techniques, and tools. In *Addison-Wesley series in computer science / World student series edition*, 1986.

[4] Domagoj Babic and Alan J. Hu. Calysto: Scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 211–220, New York, NY, USA, 2008. Association for Computing Machinery.

[5] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, page 27–38, New York, NY, USA, 2012. Association for Computing Machinery.

[6] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, 1998.

[7] Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello. Ssi properties revisited. *ACM Trans. Embed. Comput. Syst.*, 11S(1), jun 2012.

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[10] Yuandao Cai and Charles Zhang. A cocktail approach to practical call graph construction. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023.

[11] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. Finding typing compiler bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 183–198, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[13] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: Fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 499–513, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices*, 46(3):265–278, 2011.

[15] Fred Chow. Intermediate representation: The increasing significance of intermediate representations in compilers. *Queue*, 11(10):30–37, oct 2013.

[16] Cliff Click and Michael H. Paleczny. A simple graph-based intermediate representation. In *ACM SIGPLAN Workshop on Intermediate Representations*, 1995.

[17] James Cordy. The txl source transformation language. *Science of Computer Programming*, 61:190–210, 08 2006.

[18] Ronald Gary 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 Trans. Program. Lang. Syst.*, 13:451–490, 1991.

[19] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The case for virtual register machines. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '03, page 41–49, New York, NY, USA, 2003. Association for Computing Machinery.

[20] Premkumar T. Devanbu. Genoa: A customizable language- and front-end independent code analyzer. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, page 307–317, New York, NY, USA, 1992. Association for Computing Machinery.

[21] Premkumar T. Devanbu, David S. Rosenblum, and Alexander L. Wolf. Automated construction of testing and analysis tools. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, page 241–250, Washington, DC, USA, 1994. IEEE Computer Society Press.

[22] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. Windranger: A directed greybox fuzzer driven by deviation basic blocks. In *Proceedings*

of the 44th International Conference on Software Engineering, ICSE '22, page 2440–2451, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. Smoke: Scalable path-sensitive memory leak detection for millions of lines of code. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 72–82, 2019.

[24] Mathieu Fehr, Jeff Niu, River Riddle, Mehdi Amini, Zhendong Su, and Tobias Grosser. Irdl: An ir definition language for ssa compilers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 199–212, New York, NY, USA, 2022. Association for Computing Machinery.

[25] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex apis. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612. ACM, 2017.

[26] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, jul 1987.

[27] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In Yuval Yarom and Sarah Zennou, editors, *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. USENIX Association, 2020.

[28] Firefox. Firefox's building requirement on clang-11. https://firefox-source-docs.mozilla.org/build/buildsystem/toolchains.html, 2023.

[29] Jack Garzella, Marek Baranowski, Shaobo He, and Zvonimir Rakamaric. Leveraging compiler intermediate representation for multi- and cross-language verification. In Dirk Beyer and Damien Zufferey, editors, *Proceedings of the 21st International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 11990 of *Lecture Notes in Computer Science*, pages 90–111. Springer, 2020.

[30] GCC. Gcc backward compatibility. https://gcc.gnu.org/onlinedocs/libstdc++/manual/backwards.html, 2022.

[31] GCC. Gcc, the gnu compiler collection. https://gcc.gnu.org/, 2023.

[32] James Gosling. Java intermediate bytecodes: Acm sigplan workshop on intermediate representations (ir'95). *SIGPLAN Not.*, 30(3):111–118, mar 1995.

[33] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 50–61. ACM, 2011.

[34] Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. Type-directed program synthesis for restful apis. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 122–136. ACM, 2022.

[35] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.*, 4(POPL):12:1–12:28, 2020.

[36] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 27–38. ACM, 2013.

[37] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 543–553, New York, NY, USA, 2013. Association for Computing Machinery.

[38] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, page 312–326, New York, NY, USA, 2001. Association for Computing Machinery.

[39] James Hayes, William G. Griswold, and Stuart Moskovics. Component design of retargetable program analysis tools that reuse intermediate representations. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, page 356–365, New York, NY, USA, 2000. Association for Computing Machinery.

[40] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. In Longbo Huang, Anshul Gandhi, Negar Kiyavash, and Jia Wang, editors, *SIGMETRICS '21: ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, Virtual Event, China, June 14-18, 2021*, pages 81–82. ACM, 2021.

[41] Jaemin Hong and Sukyoung Ryu. Concrat: An automatic c-to-rust lock API translator for concurrent programs. *CoRR*, abs/2301.10943, 2023.

[42] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50, 2022.

[43] Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. Digging for fold: synthesis-aided API discovery for haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA):205:1–205:27, 2020.

[44] JDK. Jdk backward compatibility. https://blogs.oracle.com/java/post/upgrading-major-java-versions, 2022.

[45] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, 2019.

[46] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224. ACM, 2010.

[47] Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. Computing summaries of string loops in C for better testing and refactoring. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 874–888. ACM, 2019.

[48] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[49] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[50] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 216–226. ACM, 2014.

[51] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. Taming undefined behavior in llvm. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017,

page 633–647, New York, NY, USA, 2017. Association for Computing Machinery.

[52] Guilherme Vieira Leobas and Fernando Magno Quintão Pereira. Semiring optimizations: dynamic elision of expressions with identity and absorbing elements. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[53] Shaohua Li and Zhendong Su. Finding unstable code via compiler-driven differential testing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 238–251, New York, NY, USA, 2023. Association for Computing Machinery.

[54] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. A principled approach to selective context sensitivity for pointer analysis. *ACM Trans. Program. Lang. Syst.*, 42(2), may 2020.

[55] Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Unleashing the power of compiler intermediate representation to enhance neural program embeddings. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2253–2265, New York, NY, USA, 2022. Association for Computing Machinery.

[56] Linux. Linux kernel's building requirement on clang-11. https://www.kernel.org/doc/html/latest/process/changes.html, 2023.

[57] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[58] LLVM. Proposal of llvm-upgrader. https://releases.llvm.org/2.0/docs/CommandGuide/html/llvm-upgrade.html, 2007.

[59] LLVM. Llvm backward compatibility. https://llvm.org/docs/DeveloperPolicy.html#ir-backwards-compatibility, 2022.

[60] LLVM. The llvm compiler infrastructure. https://llvm.org/, 2023.

[61] LLVM. Llvm test guide. https://llvm.org/docs/TestingGuide.html, 2023.

[62] LLVM. Llvm's building requirement on clang-5. https://llvm.org/docs/GettingStarted.html, 2023.

[63] llvm dev. Llvm bc converter from llvm 3.9 to llvm 3.1, 2016.

[64] llvm dev. Llvm error handling mechanism, 2023.

[65] 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*, PLDI 2021, page 65–79, New York, NY, USA, 2021. Association for Computing Machinery.

[66] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Practical verification of peephole optimizations with alive. *Commun. ACM*, 61(2):84–91, jan 2018.

[67] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1867–1881, New York, NY, USA, 2019. Association for Computing Machinery.

[68] Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. Automated transpilation of imperative to functional code using neural-guided program synthesis. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022.

[69] I. A. Natour. On the control dependence in the program dependence graph. In *Proceedings of the 1988 ACM Sixteenth Annual Conference on Computer Science*, CSC '88, page 510–519, New York, NY, USA, 1988. Association for Computing Machinery.

[70] OCaml. ppxlib: Base library and tools for ppx rewriters, 2024.

[71] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. *CoRR*, abs/2203.02155, 2022.

[72] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 275–286. ACM, 2012.

[73] QT. Qt-5.12's building requirement on clang-7. https://doc.qt.io/archives/qt-5.12/supported-platforms.html, 2023.

[74] R8. R8 compiler for java. https://r8.googlesource.com/r8/+/refs/heads/main/README.md, 2022.

[75] Zvonimir Rakamarić and Michael Emmi. Smack: Decoupling source language details from verifier implementations. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 106–113. Springer, 2014.

[76] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 142–157, New York, NY, USA, 2021. Association for Computing Machinery.

[77] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[78] Rust. A pull request about upgrading llvm in rust project. https://github.com/rust-lang/rust/pull/34743, 2016.

[79] Malavika Samak, Deokhwan Kim, and Martin C. Rinard. Synthesizing replacement classes. *Proc. ACM Program. Lang.*, 4(POPL):52:1–52:33, 2020.

[80] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for c/c++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–410. Springer, 2019.

[81] Jiasi Shen and Martin C. Rinard. Using active learning to synthesize models of applications that access databases. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 269–285. ACM, 2019.

[82] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 693–706, New York, NY, USA, 2018. Association for Computing Machinery.

[83] Michelle Mills Strout, John Mellor-Crummey, and Paul Hovland. Representation-independent program analysis. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, page 67–74, New York, NY, USA, 2005. Association for Computing Machinery.

[84] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 265–266, New York, NY, USA, 2016. Association for Computing Machinery.

[85] Gil Teixeira, João Bispo, and Filipe F. Correia. Multi-language static code analysis on the lara framework. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2021, page 31–36, New York, NY, USA, 2021. Association for Computing Machinery.

[86] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. Supply-chain vulnerability elimination via active learning and regeneration. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 1755–1770. ACM, 2021.

[87] Chengpeng Wang, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang. Complexity-guided container replacement synthesis.

Proc. ACM Program. Lang., 6(OOPSLA1), apr 2022.

[88] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Enhancing cross-isa DBT through automatically learned translation rules. In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 84–97. ACM, 2018.

[89] Yichen Xie and Alexander Aiken. Scalable error detection using boolean satisfiability. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 351–363. ACM, 2005.

[90] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, jun 2011.

[91] Xiaolan Zhang, Larry Koved, Marco Pistoia, Sam Weber, Trent Jaeger, Guillaume Marceau, and Liangzhao Zeng. The case for analysis preserving language transformation. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, page 191–202, New York, NY, USA, 2006. Association for Computing Machinery.