The Journal of Systems and Software 215 (2024) 112092

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

EFACT: An External Function Auto-Completion Tool to strengthen static binary lifting☆

Yilei Zhang, Haoyu Liao, Zekun Wang, Bo Huang*, Jianmei Guo

School of Data Science and Engineering, East China Normal University, Shanghai 200062, China

ARTICLE INFO

Dataset link: https://github.com/solecnugit/EFACT

Keywords:
External function completion
Binary lifting
Static binary translation
Reverse engineering

ABSTRACT

Static binary lifting is essential in binary rewriting frameworks. Existing tools overlook the impact of External Function Completion (EXFC) in static binary lifting. EXFC recovers the declarations of External Functions (EXFs, functions defined in standard shared libraries) using only the function symbols available. Incorrect EXFC can misinterpret the source binary, or cause memory overflows in static binary translation, which eventually results in program crashes. Notably, existing tools struggle to recover the declarations of mangled EXFs originating from binaries compiled from C++. Moreover, they require time-consuming manual processing to support new libraries.

This paper presents EFACT, an External Function Auto-Completion Tool for static binary lifting. Our EXF recovery algorithm better recovers the declarations of mangled EXFs, particularly addressing the template specialization mechanism in C++. EFACT is designed as a lightweight plugin to strengthen other static binary rewriting frameworks in EXFC. Our evaluation shows that EFACT outperforms RetDec and McSema in mangled EXF recovery by 96.4% and 97.3% on SPECrate 2017.

Furthermore, we delve deeper into static binary translation and address several cross-ISA EXFC problems. When integrated with McSema, EFACT correctly translates 36.7% more benchmarks from x86-64 to x86-64 and 93.6% more from x86-64 to AArch64 than McSema alone on EEMBC.

1. Introduction

Binary lifting (Wenzl et al., 2019) is a critical process in various domains, such as binary analysis (Brumley et al., 2011; Verbeek et al., 2022; Saieva and Kaiser, 2022), binary translation (Guan et al., 2012; Fu et al., 2019; Wu et al., 2022), binary optimization (Bala et al., 2000; Dyninst Project, 2023c; Al-Tashi et al., 2019), and binary hardening (Di Federico et al., 2017). As machine code is humanly unreadable, it is essential to uplift it into a higher-level representation. Generally, there are three categories of binary lifting: binary-to-assembly (ASM), binary-to-intermediate representation (IR), and binary-to-high-level languages, e.g. C/C++.

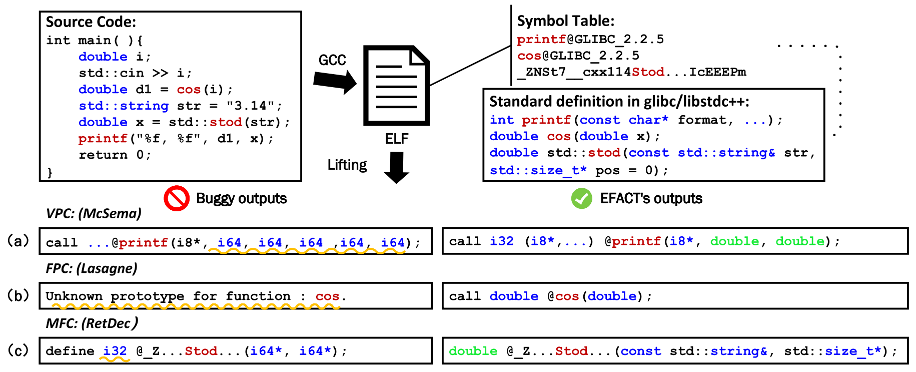
Static analysis of binaries has long been considered a challenging task, leading to the development of dynamic binary frameworks (Anand et al., 2013). However, the high cost associated with a dynamic environment makes it unsuitable for certain scenarios. In the framework of static binary lifting, the External Function Completion (EXFC) problem is inevitable when selecting IR and high-level languages as lifting outputs. An External Function (EXF) is a function defined in an external shared library, such as *glibc*. Fig. 1 shows some examples of EXFC, and let us take "*cos*" in Fig. 1 as an example. When the function named "*cos*" appears in an Executable and Linkable Format (ELF) file's (dynamically linked) symbol table, the full declaration, "*double cos (double x)*", should be returned. For binary analysis, a failure in EXFC can hinder the understanding of the binary. Meanwhile, for static binary translation, it may trigger precision errors or memory overflows, which eventually lead to severe consequences, such as program crashes.

Current static binary lifting frameworks exhibit certain deficiencies in addressing the EXFC problem, which can be broadly categorized into the following two aspects:

Deficiency 1 (***Incorrect Recovery of Function Parameters and Return Type***). Fig. 1, parts (a) and (c), evaluate the binary lifting capabilities of two state-of-the-art static binary lifting tools (Liu et al., 2022), McSema (trailofbits, 2021) and RetDec (Křoustek et al., 2017) (based on Capstone (Capstone Engine, 2023b)). We use two EXFs for evaluation, *stod* and *printf*, which correspond to mangled function completion (MFC) and variable-parameter function completion (VPC), respectively. Initially, we compile the code using the GCC compiler with the dynamic

☆ Editor: Lingxiao Jiang.
* Corresponding author.
E-mail address: bhuang@dase.ecnu.edu.cn (B. Huang).

https://doi.org/10.1016/j.jss.2024.112092
Received 16 October 2023; Received in revised form 13 April 2024; Accepted 11 May 2024
Available online 17 May 2024


Source Code:
```
int main( ){
    double i;
    std::cin >> i;
    double d1 = cos(i);
    std::string str = "3.14";
    double x = std::stod(str);
    printf("%f, %f", d1, x);
    return 0;
}
```
Symbol Table:
```
printf@GLIBC_2.2.5
cos@GLIBC_2.2.5
_ZNSt7__cxx114Stod...IcEEEPm
```
Standard definition in glibc/libstdc++:
```
int printf(const char* format, ...);
double cos(double x);
double std::stod(const std::string& str,
std::size_t* pos = 0);
```

VPC: (McSema)
```
(a) call ...@printf(i8*, i64, i64, i64 ,i64, i64);
    call i32 (i8*,...) @printf(i8*, double, double);
```
FPC: (Lasagne)
```
(b) Unknown prototype for function : cos.
    call double @cos(double);
```
MFC: (RetDec)
```
(c) define i32 @_Z...Stod...(i64*, i64*);
    double @_Z...Stod...(const std::string&, std::size_t*);
```

Fig. 1. Motivating examples of EXFC (source code is unknown in actual scenarios, we put it here to better explain the example).

linking option enabled. Subsequently, we submit the generated ELF file to McSema and RetDec. As illustrated in the bottom left corner of Fig. 1, McSema's lifting result of *printf* contains five *i64* type parameters, whereas our source code specifies two *double* type. RetDec's output indicates an *i32* as the return type of *stod*, while the correct return type is *double*. McSema and RetDec are both used as binary analysis tools, but the lifting results mislead the understanding of the source binary. This straightforward experiment illustrates that existing tools still face challenges in recovering the function declarations of variable-parameter functions and mangled functions.

Deficiency 2 (*Difficulty in Integrating a New External Library or Migrating to Another Framework*). In part (b) of Fig. 1, we evaluate the EXFC capabilities of a static binary translator, Lasagne (Rocha et al., 2022), which is based on Microsoft's llvm-mctoll (Yadavalli and Smith, 2019). For this test, we use the *cos* function, representing a simple example of fixed-parameter function completion (FPC). While Lasagne fails to recover, it is important to note that recovering *cos* alone is not a difficult task, but achieving comprehensive EXFC entails considerable effort. Lasagne mainly focuses on addressing the strong/weak memory model issue, which indeed is an important research problem in binary translation. As Lasagne researchers have not taken EXFC as part of their research, Lasagne's shortcomings in EXFC significantly diminish its overall usability. Besides, our investigation of existing open-source static binary lifting tools reveals that the source code portion responsible for handling EXFC is tightly coupled with other components. Therefore, users must possess a deep understanding of the project structure to support a new library (such as OpenSSL (OpenSSL, 2023f)) or to migrate to another binary lifting framework that currently lacks EXFC support. Although McSema and Lasagne offer an interface to manually supply the declarations of EXFs, the complexity and vastness of external libraries make this task formidable. A tool capable of auto-generating declarations can significantly enhance the accuracy and efficiency of the current static lifting process.

In response to the aforementioned limitations, we propose EFACT, an External Function Auto-Completion Tool designed for static binary lifting.

To address deficiency 1, first, we design an MFC algorithm to more accurately recover the return type of mangled EXFs in C++. While the standard LLVM's Demangling API[1] is a frequently used API for obtaining detailed function declarations of mangled EXFs, its output, as illustrated in Fig. 9(a), does not consistently guarantee the accurate identification of the function return type (Anon, 2016); only 13.5% of mangled EXFs in SPEC CPU® 2017 can get a return type through this API. Furthermore, as depicted in Fig. 1, current static binary lifting tools incorrectly treat all unknown cases as *int* (in C/C++) or *i32/i64* (in LLVM IR (Lattner and Adve, 2004)), offering no deeper recovery mechanism. Our MFC algorithm, not only recovers the implicit *this* parameter but also provides a mechanism to deal with the template specialization mechanism in C++.

Second, we develop a *Dict* auto-generator that extracts complete function declarations from external libraries and catalogs them in a dictionary (*Dict*). This not only supplements the information missing from the LLVM's Demangling API but also aids in resolving VPC and FPC issues more effectively. The lower right corner of Fig. 1 illustrates an example of EFACT's outputs, and in the motivating example, our tool outperforms three others.

Third, in the process of binary lifting, careful consideration should be taken for the input binary file, including instruction set architecture (ISA), processor architecture, operating system (OS), and library version. Existing tools overlook the impact of these dimensions in their EXFC component. Based on our *Dict* auto-generator, we construct a Library Database, which is organized layer by layer, from ISA (x86 or ARM) to *OS_Type* (CentOS or Ubuntu), and finally *OS_Version* (Ubuntu18.04 or 20.04), ensuring extensive coverage across multiple dimensions. By leveraging backward compatibility, we ingeniously cover the processor architecture and library version dimensions.

To address deficiency 2, EFACT necessitates only an ELF file as input and is capable of generating outputs in both LLVM IR and C/C++ program formats, making it an ideal plugin for binary-to-IR and binary-to-high-level language lifting frameworks. This helps enhance analysis and migration capabilities.

To make our solution more general, our tool is designed to be easily extended to embrace binaries compiled from other programming languages or binaries containing EXFs defined in other libraries. Our auto-completion workflow maximizes the utilization of the standard compiler toolchain, compilers of most programming languages generate an object file containing information akin to a symbol table, and our approach is effective as long as the "symbol table" is available.

In this paper, we select binaries compiled from C/C++ source programs along with the *glibc/libstdc++* as our framework's initial subjects, also we use binaries compiled from Fortran and Rust to test the extensibility. We choose SPEC CPU® 2017 (Standard Performance Evaluation Corporation, 2022b) as our evaluation benchmark to evaluate whether EFACT can cover 100% of the EXFs from *glibc*, *libstdc++* and *libgfortran* in it. We delve further into the static binary translation domain and evaluate whether a binary translator, augmented with EFACT, can enhance the accuracy of static binary translation. This also demonstrates the usability of our tool. Overall, our paper makes the following key contributions:

- We recognize that existing static binary lifting tools overlook the impact of failure in EXFC. To better address this problem, we introduce EFACT, an External Function Auto-Completion Tool that generates complete function declarations of EXFs presented in a target ELF file. Different from other similar tools, our tool

[1] https://llvm.org/doxygen/Demangle_8h_source.html

can recover the return type and the implicit *this* parameter of mangled EXFs based on our MFC algorithm and automatically generated dictionaries (*Dicts*) which cover popularly used external function declarations, showcasing greater robustness compared to the state-of-the-art tools, such as RetDec and McSema (Liu et al., 2022). Additionally, we delve further into the static binary translation domain, addressing several cross-ISA EXFC problems.

- EFACT is publicly available[2] and can be integrated as a plugin in other binary rewriting frameworks. We evaluate its usability with McSema and Lasagne. EFACT is easily extensible to support binaries compiled from other languages (Fortran and Rust) and binaries containing EXFs defined in other libraries (OpenSSL). Furthermore, leveraging our innovative *Dict* auto-generator, we establish a comprehensive Library Database that capitalizes on backward compatibility to encompass multiple dimensions, including ISA, processor architecture, OS, and library version.
- For SPEC CPU® 2017, EFACT achieves 100% coverage for lifting the EXFs source from *glibc*, *libstdc++* and *libgfortran*. Notably, in the context of MFC, EFACT accurately recovers 96.4% more function return types than RetDec and 97.3% more than McSema on SPECrate 2017. Furthermore, we integrate EFACT into McSema and introduce EFACT_MC, our static binary translator. This combination facilitates a 36.7% increase in successfully translating binaries from x86-64 to x86-64 and a 93.6% increase from x86-64 to AArch64 for benchmarks on EEMBC (Embedded Microprocessor Benchmark Consortium, 2023d) than the original McSema.
- We carry out a comprehensive analysis of SPEC CPU® 2017 with a focus on EXF generation. Our analysis demonstrates that factors such as ISA, optimization options, and processor architecture can impact the number of generated EXFs.

2. Background

To date, researchers have not reached a unified consensus on the output type of binary lifting, and in this paper, the terminology of EXF is different from that of the "*extern*" keyword in C/C++. In this section, we first present our understanding of binary lifting. Subsequently, we provide a comprehensive explanation of what is EXFC, and why FPC, VPC, and MFC are challenging tasks, especially in the domain of static binary analysis and static binary translation.

2.1. Binary lifting

As illustrated in Fig. 2, the process of layer-by-layer abstraction enhances the readability of the extracted information, albeit at the expense of increased abstraction complexity. While numerous researchers refer to the abstraction from binary to IR (primarily LLVM IR) as binary lifting, we align with Baldoni's perspective (Baldoni et al., 2018) that lifting is the process of transforming a more detailed representation into a more abstract one. Generally, we divide binary lifting into three categories: binary-to-ASM, binary-to-IR, and binary-to-high-level languages.

2.2. External function completion and three challenging tasks

Function completion involves recovering a detailed declaration of a function, including its return type and parameters. EXFs denote functions sourced from standard shared libraries, such as *glibc/libstdc++*. While source code contains information on parameters and return type, binary code only includes the function name in the symbol table. Recovering these function details during lifting not only enhances

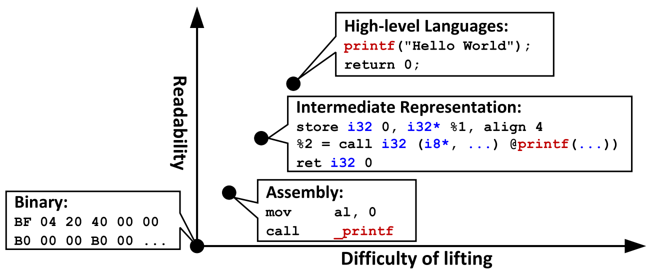


Fig. 2. Layers of binary lifting.

our understanding of the source binary but also is imperative for an IR-based or high-level language-based binary rewriting framework.

The main challenges in EXFC stem from three aspects:

FPC (fixed-parameter function completion): Functions such as *sin* and *cos* feature fixed declarations within the *glibc*, allowing us to obtain the required information directly. For FPC, the central challenge lies in developing a method that can swiftly and efficiently encompass the entire standard library.

VPC (variable-parameter function completion): A prevalent example of a variable-parameter function is *printf*, which possesses the following standard declaration:

*int printf (const char *_restrict _fmt, . . .);*

The "..." part is related to the actual binary context. Binary lifting tools need to examine the binary context before the invocation of *printf* to identify real argument types and numbers. Notably, in cross-ISA static binary translation, some detailed implementation of variable-parameter functions varies across different ISAs. For example, *va_list* is a type used in C/C++ to manage variable argument lists. Although its initialization method is identical on x86 and ARM platforms, as depicted in the declaration:

void va_start (std::va_list ap, format);

The concrete implementation within the data structure differs considerably. Fig. 4 shows an abstraction of how *va_list* is defined on x86-64 and AArch64 (with GCC 9.4, Ubuntu 20.04). This variance in implementation details leads to discrepancies in how *va_list* appears on the stack. Failure to synchronize data on the stack can result in corruption in the translated binary.

MFC (mangled function completion): Name mangling is a distinctive phenomenon in C++. Functions like "*_ZNSolsEPFRSoS_E*" may appear in the symbol table. Without demangling the function, identifying a corresponding function name in the standard library becomes virtually impossible. While the LLVM's compiler toolchain offers a Demangling API, it does not guarantee the discernment of the function's return type - only 13.5% of mangled EXFs in SPEC CPU® 2017 can get a return type through this API, leading binary lifting tools inaccurately address all unidentified cases as either *int* (in C/C++) or *i32/i64* (in LLVM IR), a flaw that can manifest in translation errors, such as casting a *double* return type to an *int*. This mismatching can potentially cause the translated program to crash during execution. The main reason stopping them from going deeper is the template specialization mechanism in C++. For instance, a preliminary analysis might reveal *_CharT* as a mangled EXF's return type from the standard library; however, determining whether to substitute it with *char16_t* or *char32_t* necessitates further investment. Overall, recovering the lost return type information, particularly for those involving template specialization, presents a considerable challenge.

3. Framework overview

In this section, we will outline the complete structure of EFACT and explain how its different parts are connected. EFACT works on the x86-64 (with Ubuntu 20.04) platform and can handle ELF files

[2] https://github.com/solecnugit/EFACT

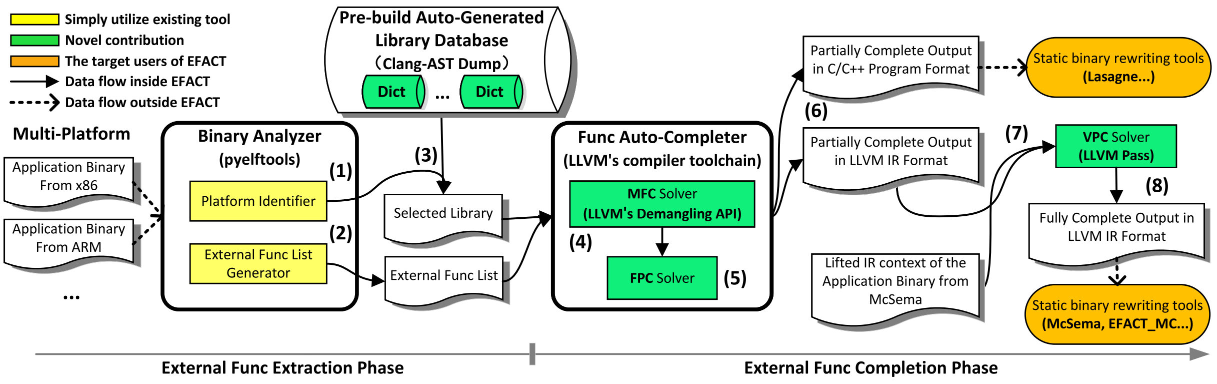

Simply utilize existing tool
Novel contribution
The target users of EFACT
Data flow inside EFACT
Data flow outside EFACT
Multi-Platform
Application Binary From x86
Application Binary From ARM
...
Binary Analyzer (pyelftools)
Platform Identifier
External Func List Generator
(1)
(2)
(3)
Pre-build Auto-Generated Library Database (Clang-AST Dump)
Dict ... Dict
Selected Library
External Func List
Func Auto-Completer (LLVM's compiler toolchain)
MFC Solver (LLVM's Demangling API)
FPC Solver
(4)
(5)
(6)
(7)
(8)
Partially Complete Output in C/C++ Program Format
Partially Complete Output in LLVM IR Format
Lifted IR context of the Application Binary from McSema
Static binary rewriting tools (Lasagne...)
VPC Solver (LLVM Pass)
Fully Complete Output in LLVM IR Format
Static binary rewriting tools (McSema, EFACT_MC...)
External Func Extraction Phase
External Func Completion Phase

Fig. 3. An overview of EFACT's auto-completion workflow (Func: Function).

```
x86-64:                                  AArch64:
struct __va_list{                        struct __va_list{
  record       __va_list_tag;              void *  __stack;
  unsigned int gp_offset;                  void *  __gr_top;
  unsigned int fp_offset;                  void *  __vr_top;
  void *       overflow_arg_area;          int     __gr_offs;
  void *       reg_save_area;              int     __vr_offs;
};                                       };
```

Fig. 4. An abstraction of the detailed implementation of *va_list* on x86-64 and AArch64 (with GCC 9.4, Ubuntu 20.04).

from different platforms, like x86-64 (with CentOS 8) or AArch64 (with Ubuntu 20.04), which is practical for real-world tasks. EFACT automatically performs a static analysis of the ELF file and provides the recovered function declarations of the EXFs within the ELF file. Fig. 3 shows the overall process of EFACT, which is divided into two main parts: the External Function Extraction Phase and the External Function Completion Phase. We will go through each step of this process in detail.

3.1. External function extraction phase

Step (1): (Section 4.1) EFACT employs *pyelftools* (Eli Bendersky, 2023g) to get the specific platform details of the source ELF file. These details include the ISA, processor architecture, OS, and information about the toolchain used to produce the binary. These details are crucial for selecting the appropriate external library versions from the Library Database. The reason why EFACT contains the step of choosing a specific version of the external library will be introduced in step (3).

Step (2): (Section 4.2) EFACT employs *pyelftools* to get the symbol table of the source ELF file. Then EFACT traverses the symbol table, extracts function symbols originating from the external standard library, and generates an EXF list. Moreover, EFACT identifies the programming language employed in the development of the source program. This step is crucial for determining the necessity of activating the MFC Solver (referenced in step (4), applicable exclusively for C++).

Step (3): (Section 4.3) Based on the platform information obtained in step (1), EFACT selects the appropriate library version and *Dict* from the Library Database. This Library Database is pre-built as preparation for practical use. The *Dict* is generated automatically, and EFACT uses a script based on Clang-AST Dump[3] to implement it. The *Dict* plays an important role in the subsequent process of function completion. Another reason why EFACT contains such a Library Database is because EFACT is designed to maximize the use of the standard compiler toolchain. While the definition of a function remains consistent across different ISAs, despite varying implementation details, using a library built for x86 to recover an ELF file from an ARM platform can lead to numerous errors. Thus, the Library Database significantly enhances the flexibility and extensibility of our tool by accommodating diverse platforms.

3.2. External function completion phase

Step (4): (Section 5.2.1) After obtaining the EXF list, EFACT deploys various strategies for function completion, taking into account the programming language used to develop the source program. As introduced in Section 2.2, three primary challenges in EXFC are FPC, MFC, and VPC. At this stage, EFACT first activates the MFC Solver if C++ is identified as the programming language of the source program. Contrasting with other static binary lifting tools that depend exclusively on the output from LLVM's Demangling API, EFACT additionally utilizes an MFC algorithm on this output. This strategy enhances EFACT's ability to more accurately recover function declarations, particularly the return types, of mangled functions.

Step (5): (Section 5.1.1) Apart from the challenges associated with name mangling in C++, EFACT's methodology for handling C++ is consistent with the approaches used for other programming languages, such as C, Fortran, and Rust. EFACT employs the FPC Solver to recover the function declarations of functions with fixed parameters.

Step (6): EFACT generates two kinds of outputs: C/C++ program code and LLVM IR. At this stage, both kinds of outputs produced by EFACT are partially complete, as they do not address the VPC challenge due to the absence of application context. Currently, the partially complete output in C/C++ program format is not designed for further exploration into VPC, as EFACT's primary focus is on binary-to-IR level static binary lifting. However, the partially complete output in C/C++ program format is already usable as input to static binary rewriting tools, like Lasagne. The partially complete output in LLVM IR format, on the other hand, will continue to undergo further recovery processes.

Step (7): (Section 5.1.2) Addressing the VPC challenge requires more than just the function declarations from the standard library. Additional information from the actual binary context is essential. EFACT's VPC Solver leverages the lifted IR context of the application binary, obtained from McSema, as a supplement to the real binary context. EFACT then employs several LLVM Passes (written by ourselves) to recover the function declarations of variable-parameter functions.

Step (8): To date, EFACT successfully generates a fully complete output in LLVM IR format. This output resolves the challenges of VPC, MFC, and FPC, and can be used for binary-to-IR level static binary rewriting tools, such as McSema and EFACT_MC.

Our tool is inspired by McSema's *generate_abi.py* and we have made EFACT publicly accessible. Our enhancements to McSema's *generate_abi.py* can be summarized as follows:

- The old ABI completion tool in McSema has remained unmaintained for an extended period, rendering it non-functional. We

[3] https://clang.llvm.org/docs/IntroductionToTheClangAST.html

rewrite this tool, making it functional and expanding its coverage beyond just the ISA perspective to encompass aspects of the OS, processor architecture, and library version.

- The old ABI completion tool in McSema requires users to supply the header files potentially utilized by the ELF file, followed by an exhaustive extraction of these files. The comprehensive extraction results are then integrated into the lifted IR, aiding McSema in more effectively lifting the source ELF file. However, it is often unclear which header files might have been used. Additionally, this extensive merging process resulted in a large volume of code in the lifted IR, despite the actual usage of only a few function declarations. In contrast, EFACT eliminates the requirement for users to provide a specific set of header files. We have developed a mechanism for extracting EXFs and automatically matching them with the corresponding function declarations in the standard library. EFACT specifically extracts the EXFs presented in the source ELF file, thereby avoiding the recovery of redundant functions. This approach is more aligned with practical scenarios and enhances usability.
- Our enhanced tool improves the auto-completion of binaries compiled from C++ programs by effectively resolving the challenges associated with C++ name mangling. Moreover, in the domain of reverse engineering, we frequently encounter ELF files that are either quite dated (developed in languages like Fortran) or extremely recent (developed in languages like Rust). We believe that a robust EXFC framework should not be limited to just C/C++, as extensibility is crucial. In comparison to the older ABI completion tool in McSema, EFACT takes a more comprehensive stance regarding language coverage. The auto-completion capabilities of EFACT have been validated with binaries compiled from Fortran and Rust source programs, which have never been tried by other researchers.
- In addition to generating LLVM IR output, our tool also generates C/C++ program output, which can be utilized by a wider range of binary rewriting tools.

4. External function extraction phase

In this section, we describe how we determine the original platform of the target ELF file and build the Library Database.

4.1. Platform identifier

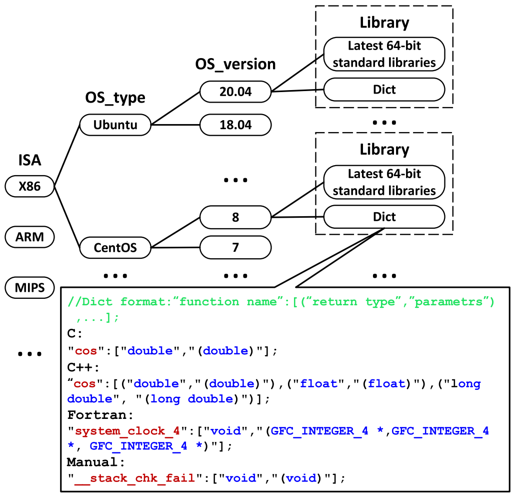
Our tool utilizes *pyelftools* to identify the ISA, processor architecture, and OS of the target ELF file. Table 1 displays the output of the motivating example in Section 1. The *Machine* and *Class* items in the ELF Header carry information about ISA and processor architecture. From the .*comment* section, we can obtain detailed information about the OS and compiler toolchain. This information assists us in locating the appropriate platform library.

4.2. External function list generator

Our tool traverses the symbol table of the target ELF file. There are two parts in the symbol table: the .*symtab* section contains all the symbols, while the .*dynsym* section contains all the dynamically linked ones. EFACT traverses both sections to cover the completable functions as much as possible. For stripped ELF, EFACT can still get the .*dynsym* section to go on our completion.

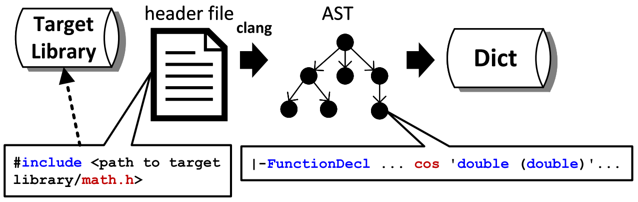
Table 1
An example of pyelftools' output.

Item	Result
Class	ELF64
Machine	Advanced Micro Devices x86-64
.comment	GCC: (Ubuntu 9.4.0-1ubuntu1 20.04.1) 9.4.0
.dynsym	cos@GLIBC_2.2.5, _ZSt3cin@GLIBCXX_3.4
.symtab	_ZNSt7__cxx114stod...IcEEEPm



```
//Dict format:"function name":[("return type","parametrs")
 ,...];
C:
"cos":["double","(double)"];
C++:
"cos":[("double","(double)"),("float","(float)"),("long
double", "(long double)")];
Fortran:
"system_clock_4":["void","(GFC_INTEGER_4 *,GFC_INTEGER_4
*, GFC_INTEGER_4 *)"];
Manual:
"__stack_chk_fail":["void","(void)"];
```

Fig. 5. Library Database framework.



```
#include <path to target
library/math.h>
```

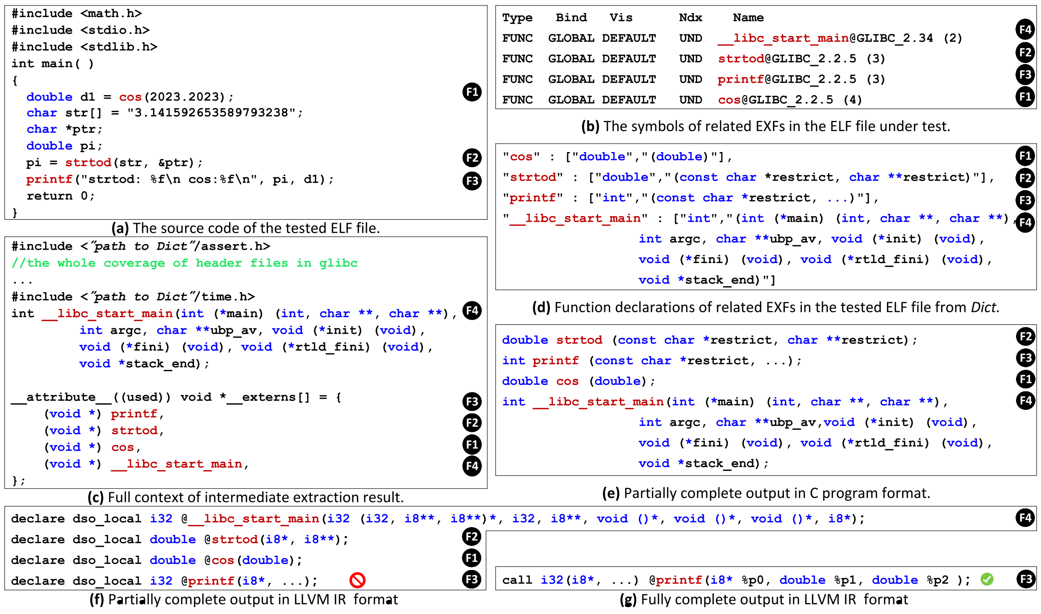
```
|-FunctionDecl ... cos 'double (double)'...
```

Fig. 6. Workflow of the *Dict* auto-generator.

4.3. Library database

In the field of reverse engineering, factors such as ISA, OS, processor architecture, and library version are critical and must be carefully considered for the input ELF. We build a Library Database to cover them. Fig. 5 shows the framework of our Library Database, organized layer by layer from ISA, to *OS_Type*, and finally *OS_Version*. We do not categorize processor architecture and library versions as separate layers because most ISAs and standard libraries generally offer backward compatibility. (ARM performs poorly in this aspect. We have tried to build a 32-bit ARM library, but the 32-bit ARM is too segmented).

There is a *Dict* in each library, which is an auto-generated file containing all the function declarations that can be detected by Clang AST-Dump in an external library. The bottom frame in Fig. 5 shows a simple example of a *Dict*. It contains the function declarations of EXFs, along with a manually added part. The *Dict* serves three important roles: First, it is the key component for EFACT in generating the output in C/C++ program format; Second, it plays a significant role in our MFC algorithm (introduced in Section 5.2.1); Third, despite our efforts to maximize the use of the standard compiler toolchain, certain exceptional cases remain unaddressed. The *Dict* can help us better address these problems. For example, in the bottom frame of Fig. 5, there is a manually added function "*_stack_chk_fail*". "*_stack_chk_fail*" is an EXF not adequately structured in the standard library, its declaration and implementation are both within a ".c" file, lacking clear separation



```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
int main( )
{
  double d1 = cos(2023.2023);                                F1
  char str[] = "3.141592653589793238";
  char *ptr;
  double pi;
  pi = strtod(str, &ptr);                                    F2
  printf("strtod: %f\n cos:%f\n", pi, d1);                   F3
  return 0;
}
```

(a) The source code of the tested ELF file.

```
Type   Bind   Vis        Ndx    Name
FUNC   GLOBAL DEFAULT    UND    __libc_start_main@GLIBC_2.34 (2)     F4
FUNC   GLOBAL DEFAULT    UND    strtod@GLIBC_2.2.5 (3)               F2
FUNC   GLOBAL DEFAULT    UND    printf@GLIBC_2.2.5 (3)               F3
FUNC   GLOBAL DEFAULT    UND    cos@GLIBC_2.2.5 (4)                  F1
```

(b) The symbols of related EXFs in the ELF file under test.

```
#include <"path to Dict"/assert.h>
//the whole coverage of header files in glibc
...
#include <"path to Dict"/time.h>
int __libc_start_main(int (*main) (int, char **, char **),          F4
         int argc, char **ubp_av, void (*init) (void),
         void (*fini) (void), void (*rtld_fini) (void),
         void *stack_end);

__attribute__((used)) void *__externs[] = {                        F3
    (void *) printf,                                                F2
    (void *) strtod,                                                F1
    (void *) cos,                                                   F4
    (void *) __libc_start_main,
};
```

(c) Full context of intermediate extraction result.

```
"cos" : ["double","(double)"],                                                  F1
"strtod" : ["double","(const char *restrict, char **restrict)"],                F2
"printf" : ["int","(const char *restrict, ...)"],                               F3
"__libc_start_main" : ["int","(int (*main) (int, char **, char **),             F4
                    int argc, char **ubp_av, void (*init) (void),
                    void (*fini) (void), void (*rtld_fini) (void),
                    void *stack_end)"]
```

(d) Function declarations of related EXFs in the tested ELF file from *Dict.*

```
double strtod (const char *restrict, char **restrict);              F2
int printf (const char *restrict, ...);                             F3
double cos (double);                                                F1
int __libc_start_main(int (*main) (int, char **, char **),          F4
                  int argc, char **ubp_av,void (*init) (void),
                  void (*fini) (void), void (*rtld_fini) (void),
                  void *stack_end);
```

(e) Partially complete output in C program format.

```
declare dso_local i32 @__libc_start_main(i32 (i32, i8**, i8**)*, i32, i8**, void ()*, void ()*, void ()*, i8*);   F4
declare dso_local double @strtod(i8*, i8**);                        F2
declare dso_local double @cos(double);                              F1
declare dso_local i32 @printf(i8*, ...);    🚫                      F3
```

(f) Partially complete output in LLVM IR format

```
call i32(i8*, ...) @printf(i8* %p0, double %p1, double %p2 ); ✓     F3
```

(g) Fully complete output in LLVM IR format

Fig. 7. An auto-completion workflow for an ELF compiled from a C program (This figure is a detailed implementation of the workflow depicted in Section 3, detailed explanation is described in Section 5.1. The source code of the binary is unknown in actual scenarios, we put it here to better explain our workflow).

(the declaration is not in a header file). Since EFACT's auto-completion methodology is traversing the function declarations in the header file suits (as introduced in Section 5), as a result, EFACT fails to detect this function. To solve this, EFACT incorporates a manual insertion approach. Manual insertion is a common method used in reverse engineering, and Box64 (ptitSeb, 2023a) also employs the same approach. Fortunately, most items appear on the *Dict* can be automatically generated with the method introduced in the next paragraph, and only rare cases trigger the manual insertion mechanism (only 1% in SPEC CPU® 2017's EXFs, building on x86-64, Ubuntu 20.04 with GCC 9.4). We take care of those rare manual insertions when building EFACT, thus it is transparent to typical EFACT users. Fig. 6 shows the workflow of the *Dict* auto-generator. First, the *Dict* auto-generator includes header files that define the target library function interfaces into a single file ("header file" in Fig. 6), then it uses Clang AST-Dump to generate the abstract syntax tree (AST) of the "header file". Afterward, the *Dict* auto-generator traverses the AST, locating the *FunctionDecl*, which refers to a function declaration or definition. Finally, the *Dict* auto-generator records all the *FunctionDecl* and gathers them in a Python *dict* class.

5. External function completion phase

After choosing a suitable library and obtaining the EXF list, EFACT moves to the completion phase. As mentioned before, there are three main challenges in the completion phase: MFC, FPC, and VPC. When dealing with FPC and VPC, binaries compiled from C/C++ source programs share the same solution approaches. MFC is a special case for C++. In this section, we will introduce the completion workflow for C and C++ separately. Our tool has two formats of completion output: C/C++ program code or LLVM IR. The output in LLVM IR format is the best-recommended format, as it covers all three challenges and maximizes the use of the Clang toolchain. Output in the C/C++ program format does not cover the VPC problem due to the lack of program context.

5.1. Auto-completion of binaries compiled from C source program

Fig. 7 shows an example of the auto-completion workflow for an ELF file compiled from a C source program. This figure is a detailed explanation of the workflow depicted in Section 3. Fig. 7(a) displays the source code of the tested ELF file. This source code is unknown in practical scenarios, we present it here to facilitate a clearer understanding of the workflow. Fig. 7(b) illustrates the symbol table of the test ELF file. This symbol table is the initial information that EFACT encounters in step (1) of Section 3. Fig. 7(c) depicts an intermediate file created by EFACT, which comprises the EXFs extracted from the tested ELF file and the *Dict* selected by EFACT to aid in the auto-completion of these EXFs. This intermediate file results from the completion of steps (1), (2), and (3) in Section 3, as input to the Func Auto-Completer. Fig. 7(d) shows the function declarations of the relevant EXFs from our chosen *Dict*. Fig. 7(e) presents the partially complete output in C program format, and Fig. 7(f) displays the partially complete output in LLVM IR format, both corresponding to the outcomes of step (6) in Section 3. Lastly, Fig. 7(g) exhibits the fully complete output in LLVM IR format, aligning with the results of step (7) in Section 3. There are four EXFs in this example, which we treat as F1, F2, F3, and F4. F1, F2, and F4 belong to the FPC problem, while F3 belongs to the VPC problem. From the source code, it is apparent that F4 is not explicitly used. This type of function, always called by the OS and inaccessible to programmers directly, is one of the functions we declared "manually" in Section 4.3.

5.1.1. FPC solver

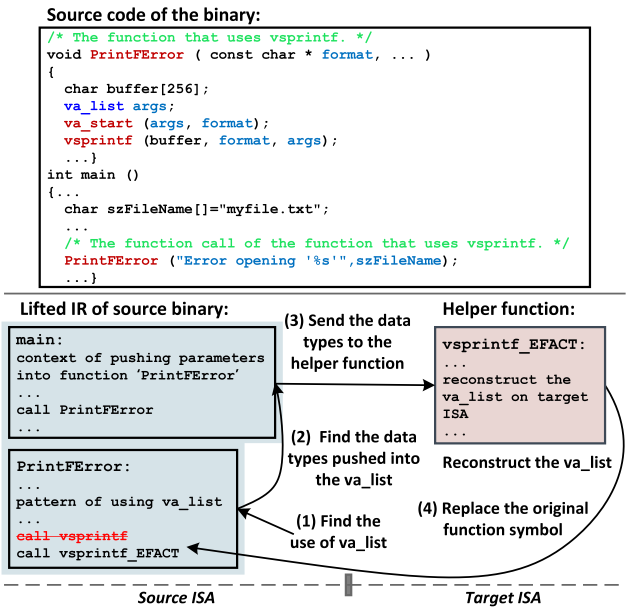
For output in LLVM IR format, our method involves putting the function name into a list, as shown in Fig. 7(c), including the target library's header file, and then passing this file to the Clang with the command ("*complete.c*" corresponds to Fig. 7(c) and "*llvm_ir_output.bc*" corresponds to Fig. 7(f)):

clang -emit-llvm -S complete.c -o llvm_ir_output.bc.

In the "*complete.c*" file, EFACT automatically adds the full declaration of "*_libc_start_main*" (as defined *manually* in Section 4.3) to avoid compile errors. Fig. 7(f) shows the recovered function declarations in LLVM IR format output from Clang, where we can see that F1, F2, and F4 have the correct declarations. However, this method cannot complete *printf*. EFACT leaves it to our VPC Solver. EFACT automatically generates the output in C program format with the information in *Dict*.

5.1.2. VPC solver

Different from fixed-parameter functions, variable-parameter functions, like *printf*, do not have a certain declaration. We see "..." (introduced in Section 2.2) in the function declaration of *printf*. These


Source code of the binary:

```
/* The function that uses vsprintf. */
void PrintFError ( const char * format, ... )
{
  char buffer[256];
  va_list args;
  va_start (args, format);
  vsprintf (buffer, format, args);
  ...}
int main ()
{...
  char szFileName[]="myfile.txt";
  ...
  /* The function call of the function that uses vsprintf. */
  PrintFError ("Error opening '%s'",szFileName);
  ...}
```

Lifted IR of source binary:

```
main:
context of pushing parameters
into function 'PrintFError'
...
call PrintFError
...
```

```
PrintFError:
...
pattern of using va_list
...
~~call vsprintf~~
call vsprintf_EFACT
```

Helper function:

```
vsprintf_EFACT:
...
reconstruct the
va_list on target
ISA
...
```

(3) Send the data types to the helper function
(2) Find the data types pushed into the va_list
(1) Find the use of va_list
Reconstruct the va_list
(4) Replace the original function symbol
Source ISA
Target ISA

Fig. 8. Workflow of cross-ISA *va_list* translation based on *Va_listPass* (source code of the binary is unknown in actual scenarios, we put it here to better explain our workflow).

parameters depend on the "*_restrict _fmt*" format passed to *printf*. From a reverse-engineering standpoint, merely having the definition available in the standard library is insufficient for function declaration recovery; the source binary's context is vital. To address this, we developed an LLVM Pass, *PrintfPass* to facilitate this functionality. This pass operates on the lifted IR context of the application binary, generated via McSema. We will introduce it based on the example in Fig. 7(a).

Firstly, *PrintfPass* locates the associated string context of "*_restrict _fmt*". As shown in Fig. 7(a), the associated string is "*strtod: %f \n cos:%f \n*". Secondly, *PrintfPass* conducts a syntactic analysis of this string. *PrintfPass* employs regular expressions to sequentially match format specifiers like "*%f*". These format specifiers within the string dictate the necessary corresponding parameters to be included as arguments in the *printf* function. Thirdly, upon matching these format specifiers, our *PrintfPass*, following the detected platform details and the standard guidelines defined in the *printf* function documentation[4], replaces the specifiers and recovers the "..." part of *printf* with the actual parameters. Fig. 7(g) displays the outcome of this process. The same approach can be easily adapted to other variable-parameter functions, such as *scanf*.

As mentioned in Section 2.2, when introducing VPC, the variable parameter represented with *va_list* needs special handling during cross-ISA static binary translation. *va_list* is a type widely used in C/C++ to manage variable argument lists, prominently utilized by functions like *vsprintf*, *vfprintf*, etc. Its initiation method is the same on x86 and ARM. However, its specific implementation on the stack differs between the two. Reconstructing the physical stack frames of the source ISA on the target ISA stands as a pivotal research problem in static binary translation. Misalignment of data on the stack can result in corrupted translated binaries. To address this, we develop another LLVM Pass named *Va_listPass*, the workflow of which is depicted in Fig. 8. To illustrate the functioning of *Va_listPass*, we use *vsprintf* as an example. The upper half of Fig. 8 demonstrates a typical usage scenario of *vsprintf* and *va_list*. The *va_list* is initialized using *va_start*. The "*format*" parameter in *va_start* and the "*args*" parameter in *vsprintf* are associated with the parameters passed from the function that utilizes them.

[4] https://cplusplus.com/reference/cstdio/printf/

The "*format*" in *vsprintf* serves a similar purpose (format specifiers, which dictate the necessary corresponding parameters to be included as arguments) as "*_restrict _fmt*" does in *printf*. To recover the *vsprintf*, *Va_listPass* undertakes the following operations:

(1) Scan the lifted IR of the source binary and locate the invocation of *va_start*. *va_start* is the detailed implementation start of *va_list*.

(2) Detect the invocation of the function that utilizes *va_list*, such as *vsprintf* here, and employs a method akin to *PrintfPass* (as introduced in the second paragraph of Section 5.1.2). This method is used to determine the data types of the parameters that are subsequently pushed into the *va_list*.

(3) Constructs a helper function on the target ISA, accepting the data types identified in step (2) as parameters. This helper function facilitates the correct reconstruction of *va_list* on the target ISA during execution.

(4) Replaces the original function call of *vsprintf* with the new helper function.

Following this process, we ensure that *va_list* can be accurately translated across multiple ISAs. Besides, *std::format* is a new function introduced in C++ 20, it is more efficient than previous string formatting options in C++, such as *vsprintf* and *printf*. With the help of the MFC algorithm mentioned in the next paragraph, the way we treat *vsprintf* and *printf* can be transplanted to *std::format*.

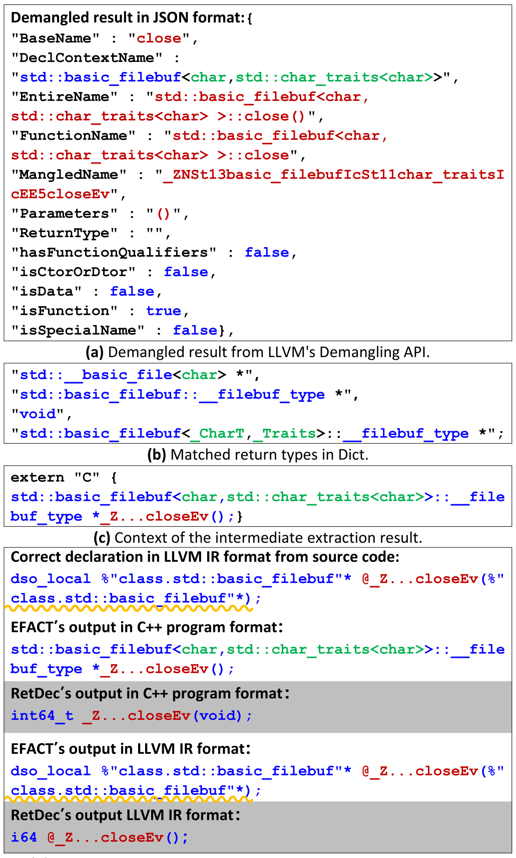
To be noticed, not all variable-parameter functions follow a format parameter similar to *printf* for determining their specific parameter lists. Attaining comprehensive coverage for all variable-parameter functions presents a considerable challenge. Based on our engineering experience, the approach to recovering variable-parameter functions often varies from one case to another. In developing the VPC Solver of EFACT, we prioritized practical applications, initially focusing on supporting commonly encountered variable-parameter functions. To elaborate, the current version of EFACT's VPC Solver is tailored based on the variable-parameter functions observed in EEMBC and SPEC CPU®2017 benchmarks. We believe that covering these benchmarks showcases the practical utility of our tool.

5.2. Auto-completion of binaries compiled from C++ source program

Fig. 9 provides a detailed breakdown of the workflow for recovering an ELF file compiled from a C++ source program, as outlined in Section 3. Apart from the challenges associated with name mangling, EFACT's methodology for handling C++ is consistent with the approaches used for C. Consequently, Fig. 9 focuses specifically on the recovery of mangled functions. Fig. 9(a) displays the output from LLVM's Demangling API. Fig. 9(b) illustrates the matched return types of the mangled function from our chosen *Dict*. Fig. 9(c) details the specific additions made to the intermediate file (as mentioned in Section 5.1) for handling mangled functions. Fig. 9(a), (b), (c) are all generated in step (4) of Section 3. Fig. 9(d) presents the output in C++ program format and LLVM IR format, corresponding to the outcomes of step (6) in Section 3. Additionally, Fig. 9(d) shows the output from RetDec for comparison. In this section, our discussion begins with an introduction to LLVM's Demangling API, followed by an explanation of how we address the name mangling problem.

5.2.1. LLVM's demangling API and MFC solver

We can see "*_ZN...@GLIBCXX_3.4*" in the symbol table of a C++ program. From the keyword "*@GLIBCXX*" we know this is a standard function defined in *libstdc++*. However, a search through the *libstdc++'s* source code fails to yield this function unless we demangle the EXF, showcasing a typical instance of C++ name mangling. Fig. 9 shows an example of how EFACT handles the mangled function. First, EFACT directly uses the LLVM's Demangling API to demangle the symbol. From Fig. 9(a), we see the output from the LLVM's Demangling API, the original function name is "*close*", which is a widely used function in C++. Second, EFACT reconstructs the demangled result to a C++



Demangled result in JSON format:
```
{
"BaseName" : "close",
"DeclContextName" :
"std::basic_filebuf<char,std::char_traits<char>>",
"EntireName" : "std::basic_filebuf<char,
std::char_traits<char> >::close()",
"FunctionName" : "std::basic_filebuf<char,
std::char_traits<char> >::close",
"MangledName" : "_ZNSt13basic_filebufIcSt11char_traitsI
cEE5closeEv",
"Parameters" : "()",
"ReturnType" : "",
"hasFunctionQualifiers" : false,
"isCtorOrDtor" : false,
"isData" : false,
"isFunction" : true,
"isSpecialName" : false},
```
(a) Demangled result from LLVM's Demangling API.

```
"std::__basic_file<char> *",
"std::basic_filebuf::__filebuf_type *",
"void",
"std::basic_filebuf<_CharT,_Traits>::__filebuf_type *";
```
(b) Matched return types in Dict.

```
extern "C" {
std::basic_filebuf<char,std::char_traits<char>>::__file
buf_type *_Z...closeEv();}
```
(c) Context of the intermediate extraction result.

Correct declaration in LLVM IR format from source code:
```
dso_local %"class.std::basic_filebuf"* @_Z...closeEv(%"
class.std::basic_filebuf"*);
```
EFACT's output in C++ program format:
```
std::basic_filebuf<char,std::char_traits<char>>::__file
buf_type *_Z...closeEv();
```
RetDec's output in C++ program format:
```
int64_t _Z...closeEv(void);
```
EFACT's output in LLVM IR format:
```
dso_local %"class.std::basic_filebuf"* @_Z...closeEv(%"
class.std::basic_filebuf"*);
```
RetDec's output LLVM IR format:
```
i64 @_Z...closeEv();
```
(d) Output respectively in C++ program format and LLVM IR format.

Fig. 9. An auto-completion workflow for an ELF compiled from a C++ program (This figure is a detailed implementation of the workflow depicted in Section 3, a detailed explanation is described in Section 5.2.)

function, shown in Fig. 9(c). We use the *extern "C"* linkage specification to let the compiler not apply name mangling to this function. Then, EFACT passes it to the compiler and gets the completed result, which is shown in Fig. 9(d).

In the portion highlighted by the yellow wavy line in Fig. 9(d), we observe an additional parameter compared to the "Parameters" obtained from the LLVM's Demangling API. This extra parameter represents the implicit *this* pointer, a fundamental element in C++. Existing tools (like RetDec in Fig. 9(d)) all omit the implicit *this* pointer in their lifting outputs of LLVM IR. They seem to overlook that the output from the LLVM's Demangling API is primarily designed for C++, and thus a more profound adaptation is necessary when lifting to LLVM IR. Omission of the *this* pointer not only misrepresents the source binary but also risks inducing parameter discrepancies during static binary translation, potentially leading to execution errors or program crashes.

From Fig. 9(a), we can see the "Return Type" of this function is empty. Although the LLVM's Demangling API reliably extracts the parameters, it does not guarantee the retrieval of the return type. Existing tools fall short of thoroughly investigating this aspect. As shown in Fig. 9(d), they simply treat all the unknown cases with an *int* (in C/C++) or *i32/i64* (in LLVM IR). This approach proves to be inadequate, especially in scenarios where the actual return type is *double*, as exemplified in Fig. 1(c). Following this procedure can result in incorrect lifting outcomes, ultimately corrupting the program after static binary translation. Besides, EFACT's output recovers more context information of the EXF than RetDec.

The main reason stopping existing tools from going deeper is the template specialization mechanism in C++. As shown in Fig. 9(b), when we try to extract the standard return type from the header

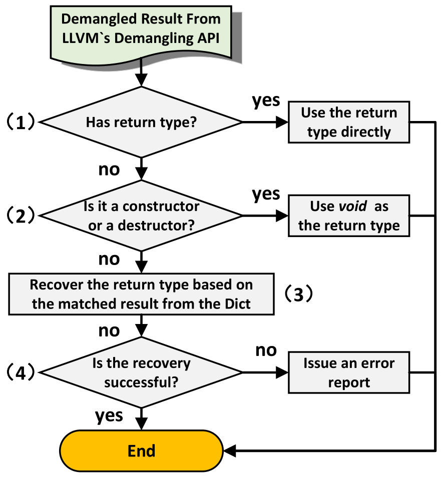

Demangled Result From LLVM`s Demangling API
(1)
Has return type?
yes
Use the return type directly
no
(2)
Is it a constructor or a destructor?
yes
Use *void* as the return type
no
Recover the return type based on the matched result from the Dict
(3)
no
(4)
Is the recovery successful?
no
Issue an error report
yes
End

Fig. 10. Workflow of the MFC algorithm.

file, we might encounter returns that are embedded with templates, necessitating further context-specific adjustments. Our MFC algorithm draws from general specifications for function declarations in C++ and implements the return value from LLVM's Demangling API. EFACT also utilizes the previously mentioned *Dict* to supplement the missing information from the LLVM's Demangling API, enhancing the accuracy and comprehensiveness of our approach. Fig. 10 illustrates the workflow of our MFC algorithm, our algorithm has the following steps:

(1) Checking the demangled result from the LLVM's Demangling API, if the "ReturnType" is not empty, we use it directly.

(2) When the "ReturnType" is empty, we inspect the "isCtorOrDtor" flag. This flag helps in identifying whether the function is a constructor or a destructor. Given that the constructor and destructor always return a *void*, we use *void* as a return type.

(3) If "ReturnType" and "isCtorOrDtor" all failed, we utilize the "BaseName" and "Parameters" retrieved from the LLVM's Demangling API, cross-verifying this data against the matched result from the *Dict*. If there is a single matched return type that does not contain templates, we select it as the result. If multiple matched returns or templates are identified, we pinpoint the correct return type based on the "DeclContextName". This item contains detailed template information, assisting us in identifying the correct results. For example, in Fig. 9(b), function "_ZNSt...closeEv" has four matched returns in *Dict*. From the "DeclContextName", we can see this function is declared from *std::basic_filebuf*, with two facilities templates. This context allows us to correctly match it with the fourth result, and replace all the templates in the result with the actual entities specified in "DeclContextName", as highlighted in the green part in Fig. 9(c).

(4) If the aforementioned steps prove unsuccessful, EFACT will issue an error report, and we recommend manually adding the function.

Fig. 9(d) shows a comparison of the recovered results from EFACT and RetDec. We can see our recovery preserves more contextual information about the original function, which aids in a more comprehensive understanding of the source binary.

5.3. Special case

Sometimes, the function symbol of an EXF might not be present in the symbol table of the source ELF file. For instance, as shown in Fig. 11, when "*cos*" is loaded using *dlopen* and *dlsym*, "*cos*" does not appear in the symbol table. The recovery process in such a case necessitates the context of the source binary. To tackle this challenge, we have developed another LLVM Pass, named *DlopenPass*. The process of *DlopenPass* involves the following steps:

```
int main() {
    void *handle;
    double (*cos_func)(double);
    ...
    /* Open the main program's symbol table. */
    handle = dlopen("libm.so.6", RTLD_LAZY);
    ...
    /* Load the 'cos' function. */
    *(void **)(&cos_func) = dlsym(handle, "cos");
     ...
     /* Use the 'cos' function. */
     printf("The cosine of 2.0 is %f\n", (*cos_func)(2.0));
     ...}
```

Fig. 11. An example of *dlopen* and *dlsym*.

Table 2
Experimental setup (GCC 9.4).

Name	OS	ISA	CPU
HD_x86	Ubuntu 20.04	x86–64	Intel® Xeon® Gold 5218R
HD_ARM	Ubuntu 20.04	AArch64	Ampere® Altra®
VM_x86_1	Centos 8	x86–64	QEMU Virtual CPU
VM_x86_2	Ubuntu 18.04	x86–64	QEMU Virtual CPU

(1) Scan the entire lifted IR of the source binary and locate all the invocations of *dlopen* and *dlsym*.

(2) Find the first parameter passed to *dlopen* to get the name of the target shared library. If this library is not included in EFACT's support libraries, generate an error report.

(3) If the target shared library is included in EFACT's support libraries, the next step is to identify the second parameter passed into *dlsym* and ascertain the corresponding function name.

(4) Add this function to the external function list and recover this function through EFACT's standard workflow.

It is important to note that the parameters examined in steps (2) and (3) may not always be constant strings. In such a case, determining the exact names of the target shared libraries or loaded functions becomes challenging. This requires additional dataflow analysis to figure out these names, thus expanding the coverage for more complicated use cases of *dlopen/dlsym*. We plan to incorporate this advanced feature in the future version of EFACT. Fortunately, our current solution can resolve many typical use cases of *dlopen/dlsym*.

6. Evaluation

6.1. Evaluation setup

To evaluate the capability of EFACT, we select five key dimensions for evaluation. In this section, we will explain the reason behind choosing each of these dimensions and describe the specific experiment designed to evaluate their respective impacts:

Coverage: This dimension is chosen to evaluate EFACT's capability to handle ELF files from diverse platforms. In reverse engineering, it is common to encounter ELF files built on numerous different platforms. This evaluation is presented as **Experiment 1** in Section 6.2.1. The methodology for experiment 1 involves using standard benchmarks to test whether EFACT can recover EXFs in these benchmarks. To simulate complex real-world workloads, we employ benchmarks from SPEC CPU® 2017. And this benchmark is constructed considering various factors, as detailed in Table 2, such as *OS_type* (*HD_x86* vs *VM_x86_1*), *OS_version* (*HD_x86* vs *VM_x86_2*) and ISA (*HD_x86* vs *HD_ARM*). The four hardware configurations listed in Table 2 are utilized to build the test benchmarks, aiming to replicate the complexity of real-world workloads as closely as possible. EFACT itself runs on *HD_x86*, which also proves its capability of lifting binaries built for different platforms.

Correctness: Merely producing a recovered output does not guarantee its accuracy. It is essential to evaluate the correctness of the tool, ensuring that the recovered results match the original function declarations in the standard library. We evaluate this dimension as **Experiment 2** in Section 6.2.2. The methodology for experiment 2 involves a manual comparison to ascertain whether EFACT's recovered results align with the corresponding declarations in the standard library. We compare EFACT's performance in this dimension with state-of-the-art static binary lifting tools like RetDec and McSema. Recognizing that manual checking is time-consuming, we opted for a smaller set of evaluation inputs compared to experiment 1. For this experiment, we selected SPEC CPU® 2017, built on *HD_x86* with *64-bit* processor architecture, and used the *O3* optimization option, to assess the correctness of EFACT's output. Experiment 2 focuses particularly on the correctness of recovery in mangled functions, as this is a key area of contribution from us.

Extensibility: As discussed in Section 3.2, a framework only for C/C++ may fall short in addressing the diverse requirements encountered in actual scenarios. EFACT is designed to support lifting binaries compiled from languages other than C/C++ and integrating libraries beyond *glibc/libstdc++*. We perform **Experiment 3** in Section 6.3.1 to evaluate this dimension. The evaluation approach of experiment 3 uses benchmarks whose source languages are not C/C++ as inputs for EFACT. Additionally, we employ an ELF file that utilizes functions from libraries other than *glibc/libstdc++* to ascertain EFACT's support for diverse libraries. We select the Fortran benchmarks from SPEC CPU® 2017, along with 10 command-line utilities written in Rust (Sam Schlinkert, 2022a), to evaluate EFACT's extensibility. As a test case for a new library, we choose *libgfortran* and OpenSSL.

Usability: As discussed in Section 1, current static binary lifting tools have limitations in solving the EXFC problem. Our goal is to enhance their capability in addressing EXFC more easily. We perform **Experiment 4** in Section 6.3.2 to evaluate the usability of EFACT. The methodology for experiment 4 involves assessing whether our tool can aid existing tools in more effectively lifting the input binary. We have chosen Lasagne and McSema as the target tools for enhancement and selected EEMBC as the evaluation input. In Section 6.3.2, we do not include the evaluation for McSema directly. Instead, we focus on EFACT_MC, a static binary translator that integrates EFACT with McSema. The evaluation of EFACT_MC can demonstrate how EFACT can assist McSema in improving binary lifting. The details of this evaluation will be elaborated in Section 6.4.

Static binary translation: We have invested a lot of effort in the EXFC required by static binary translation. To evaluate our effort, we developed a static binary translator named EFACT_MC. This tool integrates EFACT with McSema to facilitate binary translation from x86-64 to AArch64. EFACT_MC initially lifts an x86-64 binary to LLVM IR and subsequently employs the LLVM compiler to generate the target binary. We evaluate the performance of EFACT_MC in Section 6.4 as **Experiment 5**. The methodology of experiment 5 involves assessing whether EFACT_MC achieves higher translation accuracy compared to McSema alone. For this purpose, we utilize EEMBC as the evaluation input. Initially, we attempted to use SPEC CPU® 2017 as test input but encountered numerous errors (such as register mapping) while employing McSema to lift. These errors are not directly related to our primary focus, and addressing them would demand significant effort. Consequently, we use EEMBC, a benchmark that has a smaller code size but is nearly as authoritative to evaluate EFACT_MC. While the current version of EFACT_MC primarily serves to evaluate the EXFC accuracy of our framework, there is an ongoing commitment to further refine and enhance its overall performance in future developments.

Beyond the aforementioned five dimensions, we also characterize the SPEC CPU® 2017 from the perspective of EXF. This analysis is presented as **Experiment 6** in Section 6.5. The purpose of this experiment is to assist binary lifting researchers in gaining a deeper understanding of the significance of EXFC. This experiment also reveals the factors that impact the generation of EXFs.

RetDec (v5.0), Lasagne (v3[5]), and McSema (v3.0.26) are used in our experiments, and their versions are the latest ones when we conducted

[5] https://doi.org/10.5281/zenodo.6408463

Table 3
EFACT's coverage on EXFs from *glibc*, *libstdc++* and *libgfortran* in SPEC CPU® 2017 (*O3*).

Platform	EXF amount	Coverage
HD_x86	3679	100%
HD_ARM	3715	100%
VM_x86_1	3644	100%
VM_x86_2	3694	100%

*Definition of Coverage is shown in Definition 1.

our research. For all benchmarks, except those in Tables 10 and 12, we employ the *O3* optimization option. This option is the most commonly used setting in the best-known configuration (BKC) for these benchmarks. For the benchmarks in Tables 10 and 12, we use the *O0* optimization option to minimize the compiler's impact on different ISAs and processor architectures.

Current Library Database is built with the latest version (when we conducted our research) of *glibc* (v2.37), *libstdc++* (v3.4.30), *libgfortran* (v10) and OpenSSL (v3.1.1). EFACT identifies EXFs in *glibc* using the symbol "*@GLIBC*", in *libstdc++* with "*@GLIBCXX*", in *libgfortran* with "*@GFORTRAN*" and in OpenSSL with "*@OPENSSL*". We aim to expand the range of symbols recognized in future versions of EFACT. SPEC CPU®2017 is organized into 4 suites: SPECrate 2017 Integer (intrate), SPECspeed 2017 Integer (intspeed), SPECrate 2017 Floating Point(fprate), and SPECspeed 2017 Floating Point (fpspeed).

6.2. Coverage and correctness

6.2.1. Coverage

We use SPEC CPU®2017 to evaluate the coverage of our tool. The ability of coverage is defined as follows:

Definition 1. Coverage = (Number of EXFs can be completed by EFACT) / (Number of EXFs in the target ELF file from *glibc*, *libstdc++* and *libgfortran*)

Table 3 shows the coverage of EFACT. From the "EXF amount" column, we can see the platform does impact the generation of EXFs. First, when examining different ISAs, we observe that the EXF amount on *HD_ARM* exceeds that on *HD_x86*. This difference is attributed to ARM's weak memory model and RISC architecture, which we have detailed discussed in Section 6.5. Second, when facing different OS, the EXF amounts on *Ubuntu 20.04* (*HD_x86*) and *Ubuntu 18.04* (*VM_x86_2*) are higher than *CentOS 8 (VM_x86_1*). On Ubuntu systems, compiling with the *O3* optimization option triggers the "_FORTIFY_SOURCE" macro to be set to "2", leading to the generation of many EXFs for buffer overflow detection. Conversely, on *CentOS 8*, the "_FORTIFY_SOURCE" macro is not defined, and the compiler does not enforce buffer overflow detection. Concerning different versions of the same OS, we note that the EXF amount on *VM_x86_2* is bigger than the amount on *HD_x86*, with the additional EXFs being functions related to buffer overflow detection. EFACT can achieve 100% coverage on the EXFs from *glibc*, *libstdc++*, and *libgfortran* of all benchmarks, not restricted by the platform.

6.2.2. Correctness

We delve deeper into the recovery of SPEC CPU® 2017 on *HD_x86*. First, we introduce the detailed performance of each part in EFACT. Table 4 shows the distribution of VPC, FPC and MFC of each suite in SPEC CPU® 2017. In the "this" column of Table 4, 69.5% mangled EXFs in SPEC CPU® 2017 will pass an implicit *this* parameter, while all existing tools overlook to recover it. In the "has ret" column, only 13.5% mangled EXFs can get a return type from the LLVM's Demangling API.

Second, we perform an overall comparison between the correctness of the output from McSema, RetDec, and EFACT. Table 5 shows the experiment result. Correctness is defined as follows:

Table 4
EFACT's detailed analysis result on EXFs from *glibc*, *libstdc++* and *libgfortran* in SPEC CPU® 2017 (*HD_x86*, *64-bit*, *O3*).

Suite	EXFs	VPC	FPC	MFC		
				num	this	has ret
intspeed	804	51	593	160	119	21
intrate	800	51	591	158	117	18
fpspeed	847	43	760	44	29	5
fprate	1228	62	993	173	107	28
Ratio					69.5%	13.5%

***this**: num of functions which has a "this" pointer as para- meter. **has ret:** num of demangled outputs from LLVM's Demangling API that has a return type. SPEC CPU® 20- 17 is organized into 4 suites. **MFC num** ≠ **this** + **has ret**.

Table 5
Comparison among RetDec, McSema, and EFACT on the correctness of EXFC from *glibc*, *libstdc++* and *libgfortran* on SPEC CPU® 2017 (LLVM IR format, *HD_x86*, *O3*, *64-bit*).

Function type	EXF	Correctness		
	amount	McSema	RetDec	EFACT
variable parameter	120	0%	100%	100%
fixed parameter	1612	4.2%	89.7%	100%
mangled	444	1.3%	4.7%	100%

*Definition of Correctness is shown in Definition 2.

Definition 2. Correctness = (Number of completed EXFs that match the function declarations in the standard library) / (Number of EXFs in the target ELF file from *glibc*, *libstdc++* and *libgfortran*)

To be noticed, the EXF amount in Table 5 is smaller than that in Table 4. This discrepancy is because, during our evaluation, only 34 of the total 43 benchmarks in SPEC CPU®2017 could be successfully analyzed by all three tools. Table 5 reflects the evaluation results for these 34 benchmarks. In the "variable parameter" row, both RetDec and EFACT achieve 100%. RetDec employs a recovery methodology similar to ours on recovering the variable-parameter function, but it does not consider the cross-ISA situation. McSema gets 0%, indicating it does not address the VPC challenge; In the "fixed parameter" row, RetDec gets 89.7%, performing well on EXFs from *glibc* and *libsdc++* but lacking coverage for the *libfgortran* library. RetDec incorrectly recovers all the EXFs from *libfgortran* library. McSema only gets 4.2% on the FPC, treating all argument types as *i64*, regardless of their actual types like a pointer or *double*, which significantly reduces recovery accuracy. EFACT gets 100% in this category; In the "mangled" row, both McSema and RetDec get a low percentage, with only 1.3% and 4.7% respectively. This category is a major contribution of our work. To further validate our tool's advancements in this respect, we have conducted a more detailed experiment in the next paragraph.

We evaluate the capabilities concerning the return-type recovery of mangled EXFs. In Table 6, we choose four dimensions to test the ability of mangled EXF recovery. "AC" represents whether the recovered return type is the same as those from the standard library; "info loss" means the recovered return type lacks context info, which is mentioned in Section 5.2.1; "crash" indicates that the recovered return type may cause the program to crash after static binary translation. Crash is often triggered by casting a data type with more bits to a data type with less bits, which is mentioned in Section 2.2. "widening" refers to scenarios when casting a data type with less bits to a data type with more bits. An example of this is when a function with a true return type of *i32* is cast to a return type of *i64*. In static binary translation, assigning an *i32* to *i64* does not induce data loss, but may affect the execution efficiency of the program after translation, as this might impact register allocation and memory layout.

Table 6 delineates a comparative analysis among RetDec, McSema, and EFACT on SPECrate 2017. From this table, we can see our tool performs best. A notable discrepancy is observed in the "AC" column for each of the tools, in which EFACT exhibits a remarkable

Table 6
Comparison among RetDec, McSema, and EFACT on the ability of MFC on SPECrate 2017 (LLVM IR format, *HD_x86*, *O3*, *64-bit*).

Benchmark	**EXF**	**info**	**RetDec**				**McSema**				**EFACT**			
	num	**included**	**AC**	**crash**	**widening**	**info loss**	**AC**	**crash**	**widening**	**info loss**	**AC**	**crash**	**widening**	**info loss**
leela_r	53	25	2	1	50	25	1	0	52	25	53	0	0	0
xalancbmk_r	31	13	0	0	31	13	0	0	31	13	31	0	0	0
omnetpp_r	76	35	6	6	64	35	4	2	70	35	76	0	0	0
cactuBSSN_r	44	17	1	1	42	17	1	0	43	17	44	0	0	0
namd_r	6	0	0	0	6	0	0	0	6	0	6	0	0	0
parest_r	101	51	3	4	94	51	3	1	97	51	101	0	0	0
povray_r	4	0	0	0	4	0	0	0	4	0	4	0	0	0
blender_r	18	9	0	0	18	9	0	0	18	9	18	0	0	0
Sum	333	150	12	(12) 4/8	309	150	9	(3) 2/8	321	150	333	(0) 0/8	0	0
Improvement			96.4%	50%	92.8%	100%	97.3%	25%	96.4%	100%				

info included: num of functions that have detailed context info in the standard library;* ***AC:* *num of recovered return types which are the same as those from the standard libr- ary;* ***crash:*** *num of recovered return types which may cause the program to crash after static binary translation. In the* ***Sum*** *raw of* ***crash****, "()" refers to the total num of recov- ered return types that can lead to a crash. We also present the crash ratio of all tested benchmarks.* ***Improvement****: shows the improvement of EFACT compared to theirs.* ***EXF num*** *=* ***AC*** *+* ***crash*** *+* ***widening****.*

improvement, accurately recovering 96.4% more function return types than RetDec and 97.3% more than McSema. This is mainly due to the simplistic method employed by RetDec and McSema in recovering the unknown return type with *i32* and *i64*, ignoring the fact that numerous mangled EXFs in SPECrate 2017 return a *void*, *double* or a pointer. RetDec and McSema have the same ratio of "info loss", as their recovery process does not concern the declaration context inherent to the source function declaration. The improvement of EFACT on "info loss" is 100%. Besides, McSema experiences fewer crashes compared to RetDec, a consequence of the broader acceptance range of the *i64* return type. RetDec may lose data when casting an *i64* return type to *i32*. Relatively, this policy also causes McSema to be more likely to trigger type widening. Based on our MFC Algorithm, our tool outperformed the other two in all aspects.

6.3. Extensibility and usability

6.3.1. Extensibility

This paper mainly introduces how we implemented the lifting for EXFC on *glibc/libstdc++* in EFACT, which performs exceptionally well. In fact, our approach can be easily extended to support the lifting for binaries containing invocations to EXFs defined in other libraries, and binaries compiled from other programming languages.

To support an additional library, such as OpenSSL, we first identify the symbol that can be used to extract EXFs. In the ELF file using OpenSSL, symbols like "*@OPENSSL*" are generated. We use this symbol to extract the EXF list. Second, we utilize the *Dict* auto-generator introduced in Section 4.3 to construct a *Dict* for OpenSSL. Finally, based on the EXF list and the newly generated *Dict*, we follow the same workflow to complete the process. According to our evaluation, EFACT shows 100% coverage on EXFs of the OpenSSL executable (which provides a command-line interface) in the OpenSSL library.

Although there may be significant differences between languages, the *compile-link-and-run* chain is largely the same. The compiler of most languages generates an object file containing information akin to a symbol table. Our approach is effective as long as the "symbol table" is available. Our tool also supports recovering the *libgfortran* library used in Fortran. We achieved 100% coverage on the Fortran benchmarks in SPEC CPU® 2017. We also chose 10 command line utilities[6] written in Rust. EFACT achieved a 100% recovery rate for the EXFs from *glibc*, which is used in the Rust utilities.

[6] bat, dust, fd, fend, hyperfine, miniserve, ripgrep, just, cargo-audit and cargo-wipe.

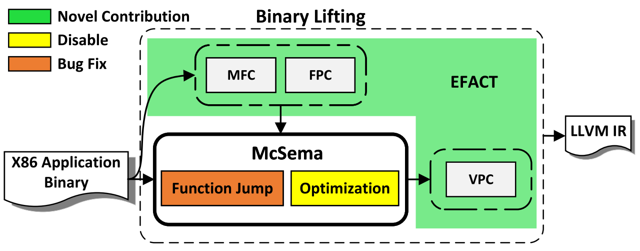


Fig. 12. EFACT_MC lifting workflow.

6.3.2. Usability

As mentioned in Section 1, to reduce the investment of binary lifting researchers in solving EXFC, we designed EFACT as a lightweight plugin. Our trials across various existing tools confirm that EFACT can adeptly aid them in resolving the EXFC problem.

Lasagne: In Fig. 1, we can see Lasagne performs poorly in EXFC. To assess the improvement in Lasagne when combined with EFACT, we first use a binary compiled from the source code in Fig. 1 as a test input. With EFACT's assistance, Lasagne successfully lifted the binary into LLVM IR. Second, we tested a handmade binary containing 20 EXFs, and Lasagne successfully lifted this binary as well. Third, for a more robust evaluation, we use the EEMBC benchmarks as test inputs. In this evaluation, EFACT's supplement helped Lasagne avoid the "Unknown prototype for function" error, but Lasagne failed in the subsequent instruction semantic translation step due to its lack of instruction coverage. Although Lasagne (combined with EFACT) could not lift the EEMBC benchmark completely, EFACT indeed helped Lasagne progress further in the binary lifting process.

McSema: McSema is a well-known static binary lifting tool, which features an interface (*–abi_loader*) to accommodate function declarations in LLVM IR format. The EFACT output in LLVM IR format integrates well with this interface. To evaluate the improvements when combining McSema with EFACT, we developed a static binary translator, EFACT_MC, which integrates McSema and EFACT. The framework of EFACT_MC is shown in Fig. 12. We chose the EEMBC benchmarks to validate EFACT_MC, using the original McSema for comparison. The specific details of this experiment are described in Section 6.4.

6.4. Static binary translation

Besides the above four dimensions, our tool has done a lot of work on solving the EXFC in the static binary translation area.

To evaluate our work, we employ an x86-64 to AArch64 (on Ubuntu 20.04) static binary translator, EFACT_MC, which is based on McSema and EFACT. We use 4 benchmarks of EEMBC (build on *HD_x86* with Clang11) as input binaries. Table 7 provides a brief introduction to them (other benchmarks require mobile hardware for installation or

Table 7
EEMBC benchmark brief introduction.

Benchmark	Introduction
CoreMark_Pro 1.0	Sophisticated test of a processor's functionality
AutoBench 2.0	Multicore processor automotive workloads
MultiBench 1.1	Multicore processor integer workloads
FPMark 1.0	Multicore processor floating-point workloads

are not buildable due to insufficient documentation support). Our evaluation method consists of the following steps: First, we build the EEMBC benchmark and obtain the test binary (on *HD_x86*). Second, we put the binary into both McSema and EFACT_MC, obtaining the lifted LLVM IR (on *HD_x86*). Then, we use Clang11 to compile the lifted LLVM IR to ELF (on *HD_ARM* and *HD_x86*). Finally, we test the correctness of recompiled ELF from McSema and EFACT_MC, respectively. All EEMBC benchmarks output their execution results to the terminal. We compare these results (outputs from the EEMBC benchmarks compiled from source code) with those from EFACT_MC and McSema to verify the correctness of the translated binary.

Fig. 12 shows an overview of the binary lifting process facilitated by EFACT_MC. We use EFACT as a plugin to help McSema better solve the EXFC problem. To minimize the potential impact from other factors, we fix several bugs in the function jump part of McSema, and disable all the default optimization options to avoid potential interfering factors. We also apply the above modifications to the McSema which is used as a comparison baseline.

Tables 8 and 9 show the evaluation result. We categorize four evaluation indexes: Pass, ①, ②, and ③. ①mainly results from McSema's ability in control-flow-graph recovery or x86-64 instruction coverage, which is not related to the EXFC problem. ②and ③are strongly related to our focus because incorrect parameter passing to the dynamically linked shared library or a wrong return type from the shared library can both cause the translated program to crash or encounter errors.

We first compare the results that choose x86-64 as the target ISA. The goal of this series of experiments is to sidestep the differences between ISAs, allowing us to more accurately gauge the enhancements that complete function declaration recovery can introduce to static binary translation. From Table 8, we can see EFACT_MC successfully translates 36.7% more benchmarks than McSema (without EFACT). Particularly, within the confines of the green rectangle in AutoBench, our tool has been instrumental in solving program crashes, proving our earlier conclusion that the EXFC problem can indeed lead to a program crash. For FPmark, there is a significant improvement in the translation rate. FPmark is a floating-point workload, it contains many functions that use *double* as return type and parameters. The default EXFC policy employed by McSema—completing *double* with *int*—triggers serious problems.

Then we compare the results that choose AArch64 as the target ISA, where we intend to highlight that the differences between ISAs necessitate considerable modifications, as showcased in this series of tests. From Table 9, we can see McSema failed to translate all the benchmarks, This is because the source code of EEMBC uses a lot of *va_list* to display messages. However, with the help of our LLVM Passes mentioned in Section 5.1.2, EFACT_MC successfully translates 93.6% more benchmarks than McSema.

From the column ①in Tables 8 and 9, EFACT_MC shows no improvement. All seven of these failed cases report errors due to encountering unsupported x86-64 instructions while lifting the source ELF. These errors are unrelated to EXFC. In future developments, we aim to improve EFACT_MC's coverage on x86-64 instructions.

Overall, EFACT greatly improves the accuracy of static binary translation.

6.5. EXF's perspective on SPEC CPU® 2017

To help binary lifting researchers better understand the EXFC problem, we have analyzed all the EXFs in SPEC CPU® 2017, without library restrictions, and we chose ISA, optimization option, and processor architecture (three frequently concerned aspects in binary lifting) for characterization. It should be noted that the counts of EXFs (on *HD_x86*, 64-bit) in Table 10, Table 11, and Table 12 exceed those in Table 4. While Table 4 displays EXFs souring from *glibc*, *libstdc++* and *libgfortran*, the latter three tables count all EXFs in the benchmark. This includes libraries beyond the aforementioned three. For instance, *620.omnetpp_s* contain EXFs sourced from "*@OMP*".

First of all, from Table 10, we can see there is a significant amount of EXFs that cannot be ignored. The number of EXFs in *intspeed* is nearly the same as *intrate* because these two suites have the same benchmark numbers and application areas, but differ in their performance calculation methods. The small difference mainly lies in their different output format (e.g. *505.mcf_r* use more output functions than *605.mcf_s*, like *fprintf*) and their measurement methods (e.g. *657.xz_s* use more functions to get measurement matrix than *557.xz_r*, like *omp_get_max_threads*). *fpspeed* and *fprate* differ so much because *fprate* has 3 more benchmarks than *fpspeed*.

Second, we observe that different ISAs impact the generated EXF numbers. *HD_ARM* has 1.5% more EXFs than *HD_x86*. Upon further investigation, we found that *HD_ARM* requires an additional *abort* function in each generated ELF compared to those on *HD_x86*. This is because AArch64 employs an *abort* function to handle scenarios where the main program (or *_libc_start_main*) returns unexpectedly. In contrast, x86-64 uses the *HLT* instruction for this purpose. Furthermore, *HD_ARM* requires more functions to execute the same floating-point operations than x86-64 (owing to AArch64's RISC architecture).

Third, from Table 11, we can see the optimization option also impacts the number of generated EXFs. For example, from *intspeed*, *intrate* and *fprate* of Table 11, we can see sometimes a higher optimization option decreases the number of generated EXFs. For example, when compiling *657.xz_s* with the *O3* optimization option, it will result in the inlining of the *strlen* function, thereby reducing the number of EXFs. However, it is not always accurate to state that a higher compiler optimization option leads to fewer EXFs. Table 11 shows an increase in EXFs under the *O3* optimization option compared to *O0* for *fpspeed*. Further analysis shows that compiling *621.wrf_s* with the *O3* optimization option causes more EXFs to be generated, including a group of functions like *_ZGVdN4v_sin* from *Libmvec*. *Libmvec* is an x86-64 *glibc* library incorporating SSE4, AVX, AVX2, and AVX-512 vectorized functions for mathematical operations like *cos*, *exp*, *sin*, etc. On the other hand, when *621.wrf_s* is compiled with the *O0* optimization option, the compiler will not enable the Vectorization optimization, therefore, the generated binary will not include EXFs from *Libmvec*, leading to a reduced number of EXFs.

Fourth, Table 12 shows the EXF numbers for both 32-bit and 64-bit benchmarks. Only the SPECrate suites support *-m32* to generate 32-bit ELF files[7]. We conduct this experiment on *HD_x86*. From the table, we can see the 32-bit benchmarks have 0.5% more EXFs than the 64-bit benchmarks. When building the 64-bit benchmarks, the compiler usually links with newer library versions than the 32-bit ones. Newer version libraries have some default optimizations that may inline parts of the internal implementation of certain functions (transparent to the programmer) to make the program faster, resulting in fewer EXFs.

In summary, our analysis confirms that EXFC is an inevitable challenge. Factors such as ISA, optimization option, and processor architecture directly impact the number of generated EXFs. Our study reveals that in SPEC CPU®2017, AArch64 binaries have 1.5% more EXFs than x86-64 binaries for SPEC CPU® 2017, due to its weak memory

[7] https://www.spec.org/cpu2017/Docs/overview.html

Table 8
EFACT_MC's evaluation result (x86-64 to x86-64).

Benchmark	Test case amount	McSema				EFACT_MC				Improvement
		Pass	①	②	③	Pass	①	②	③	
CoreMark_Pro	9	6	1	0	2	8	1	0	0	22.2%
AutoBench	20	12	0	2	6	20	0	0	0	40.0%
MultiBench	28	23	0	0	5	28	0	0	0	17.9%
FPmark	52	21	6	0	25	46	6	0	0	48.1%
Sum	109	62	7	2	38	102	7	0	0	36.7%

**Pass: successfully run without bugs. ①: fail to be recompiled and linked to an ELF. ②: suc- cessfully generate the ELF, but some bugs crash the program while running. ③: successfully generate the ELF, but have some bugs that do not crash the program while running.*

Table 9
EFACT_MC's evaluation result (x86-64 to AArch64).

Benchmark	Test case amount	McSema				EFACT_MC				Improvement
		Pass	①	②	③	Pass	①	②	③	
CoreMark_Pro	9	0	1	8	0	8	1	0	0	88.9%
AutoBench	20	0	0	20	0	20	0	0	0	100%
MultiBench	28	0	0	28	0	28	0	0	0	100%
FPmark	52	0	6	46	0	46	6	0	0	88.5%
Sum	109	0	7	102	0	102	7	0	0	93.6%

**Pass: successfully run without bugs. ①: fail to be recompiled and linked to an ELF. ②: suc- cessfully generate the ELF, but some bugs crash the program while running. ③: successfully generate the ELF, but have some bugs that do not crash the program while running.*

Table 10
Total EXF counts of SPEC CPU® 2017 on different ISA (*64-bit ,O0*).

Suite	HD_x86	HD_ARM	difference
intspeed	931	939	0.9%
intrate	923	934	1.2%
fpspeed	1004	1016	1.2%
fprate	1374	1407	2.4%
Sum	4232	4296	1.5%

***difference**: how many EXFs HD_ARM have than HD_x86.

Table 11
Total EXF counts of SPEC CPU® 2017 with different optimization option (*HD_x86, 64-bit*).

Suite	O0	O3	difference
intspeed	931	859	−7.7%
intrate	923	853	−7.6%
fpspeed	1004	1043	3.9%
fprate	1374	1314	−4.4%
Sum	4232	4069	−3.9%

***difference**: how many EXFs O3 have than O0.

Table 12
Total EXF counts of SPEC CPU® 2017 on different process architecture (*HD_x86 ,O0*).

Suite	32-bit	64-bit	difference
intspeed	–	931	–
intrate	923	923	0%
fpspeed	–	1004	–
fprate	1385	1374	−0.8%
Sum	2308	2297	−0.5%

*raw **Sum** calculate intrate and fprate. **difference:** how many EXFs 64-bit have than 32-bit.

model and RISC architecture; 32-bit benchmarks have 0.5% more EXFs than 64-bit benchmarks as 64-bit benchmarks link to newer version libraries, which likely inline some implementation functions, resulting in fewer EXFs. We recommend that binary lifting researchers focusing on the ARM platform and 32-bit processor architecture should pay more attention to EXFC.

7. Limitations

The current version of EFACT has several limitations:

(1) EFACT cannot support the shared library if the library's source files, in which the functions are declared, are missing.

(2) Extra manual effort is needed for a few libraries if their source files, in which the functions are declared, are not well organized, like *libgfortran*. We have automated much of this process through scripts.

(3) Although EFACT demonstrates 100% coverage in the experiments described in Section 6.2.1, it is important to note that our focus has been on covering benchmarks like EEMBC and SPEC CPU®2017. While achieving full coverage on these benchmarks suggests practical usability, there may still exist functions, in GNU standard libraries, that EFACT fails to recover. EFACT provides a manual insertion interface, allowing users to add EXFs that the current version may not support. With more practical workloads explored by us using EFACT, its coverage on EXFC will be continuously improved accordingly.

(4) Our current LLVM Passes have only been verified for compatibility with the LLVM IR lifted by McSema. Our VPC solver is not guaranteed to recover all the variable-parameter functions. Supporting the EXPC for more functions with variable parameters is one of our future plans.

8. Related work

In this section, we first discuss function signature recovery. Then, we discuss dynamic binary lifting. Finally, we shift our focus to the related works of EFACT and divide them into three main categories based on output format: binary-to-ASM, binary-to-IR, and binary-to-high-level languages.

8.1. Function signature recovery

Function signature recovery (van der Veen et al., 2016; Muntean et al., 2018; Lin and Gao, 2021) aims to recover the number and types of arguments of a function from binary executables, where "arguments" refer to the function's parameters, which does not include the return type. Different from EFACT's application domain, function signature recovery is used to construct a fine-grained control-flow graph (CFG) for aiding control-flow integrity (CFI) enforcement. It operates at the binary-to-ASM level and is used for defending against control-flow hijacking attacks, ensuring that control flows only occur between callees and callers with matching function signatures.

Table 13
Related work overview (static phase).

Tool	Output	EXFC		
		FPC	VPC	MFC
EFACT	C/C++,LLVM IR	✓	✓	✓
McSema	LLVM IR	✗	✗	✗
SSM	ASM	×	×	×
SmartDec	C++	×	×	×
Ghidra	C/C++,ASM	✓	✗	✗
RetDec	C,LLVM IR	✓	✗	✗
Lasagne	LLVM IR	×	✗	×
IDA PRO	C/C++,ASM	✓	✗	✗

*✓: correctly recover EXF and match all the require- ments of the challenge. ✗: can generate a recovered result, but failed to match all the response challenge' requirements. × :cannot generate a recovered result.

8.2. Dynamic binary lifting

Before 2020, binary lifting primarily appeared in static frameworks. Although many dynamic frameworks at that time (Guan et al., 2008; Hong et al., 2012; Rokicki et al., 2019) had processes for translating binaries into IR, researchers did not specifically propose the concept of dynamic binary lifting due to the presence of interpreters in dynamic frameworks. In 2020, BinRec (Altinay et al., 2020) introduced the concept of Dynamic Binary Lifting, presenting a detailed analysis of how dynamic environments could address the issues present in previous static binary lifting methods.

8.3. Static binary lifting tools

Table 13 presents the summary of existing tools' capabilities in addressing the EXFC problem. We will present an overview on each of them.

8.3.1. Binary-to-ASM

ASM can accurately represent the behavior of the source binary. However, ASM lacks direct information on function parameters and return type, making it difficult to supplement the details of function calls. IDA PRO (Hex-Rays, 2014) and Ghidra (National Security Agency, 2023e) are two well-established tools that not only output the source binary's ASM but also provide a transformation interpretation in C/C++ program format. There are still shortcomings when applied to the EXFC issue, especially the return type of mangled EXF.

8.3.2. Binary-to-IR

Many static binary frameworks utilize LLVM IR or TCG IR (Bellard, 2005) for translation or analysis, which is highly relevant to our work and has significant potential benefits. After trying to obtain code links and contacting corresponding authors of relevant papers via email, both LLBT (Shen et al., 2012) and SecondWrite (Anand et al., 2013) were unable to reproduce their work. Moreover, neither of their papers mentioned the EXFC issue. McSema and RetDec, as mentioned in Section 1, they have respective shortcomings. Lasagne experienced significant problems with FPC and MFC. Despite adding *cmath.h* to the "*–include-files*" option, Lasagne is still unable to lift *cos*. Incorrect supplementation of function parameters during static binary translation can result in parameter loss or type errors, potentially leading to a program crash when translating complex software. Existing tools all failed to recover the implicit *this* parameter in MFC. Besides our tool, RetDec performs best in this area. However, it cannot provide the accurate return type for mangled EXF.

8.3.3. Binary-to-high-level languages

Few tools are capable of producing quality high-level languages such as C and C++, as doing so requires a significant amount of information and engineering effort. IDA PRO and Ghidra generate the most accurate output in C format. Phoenix (Brumley et al., 2013), FoxDec (Verbeek et al., 2020), and SmartDec (Fokin et al., 2011) (C++ format) are three other published lifting tools. Unfortunately, these tools have not discussed the EXFC problem in their papers. Additionally, Phoenix is not an open-source tool, and FoxDec's module responsible for generating C language output remains closed-source. SmartDec has shown poor performance on EXFC.

9. Conclusion

In this paper, we present EFACT, an External Function Auto-Completion Tool for static binary lifting frameworks. EFACT can recover the full function declarations of an ELF'EXFs, unbounded by ISA, processor architecture, OS, and library version. Notably, our tool can recover the implicit *this* parameter and return type of mangled EXF with the help of our MFC algorithm. EFACT automatically generates dictionaries (*Dicts*) to cover popularly used external function declarations.

EFACT demonstrates 100% coverage of EXFs derived from the *glibc*, *libstdc++* and *libgfortran* on SPEC CPU® 2017. Notably, in the context of MFC, EFACT exhibits a remarkable improvement, accurately recovering 96.4% more function return type than RetDec and 97.3% more than McSema on SPECrate 2017. EFACT generates output in C/C++ program format and LLVM IR format to better assist other binary rewriting frameworks. It has been tested and proven compatible with Lasagne and McSema.

Additionally, we delve further into the static binary translation domain, addressing several cross-ISA EXFC challenges. We built a static binary translator called EFACT_MC, which integrates EFACT into McSema and translates binary from x86-64 to AArch64 (on Ubuntu 20.04). EFACT_MC's performance evaluation showed a notable increase in translation accuracy, with 36.7% more correct translations in EEMBC benchmarks when moving from x86-64 to x86-64, and 93.6% more from x86-64 to AArch64, compared to the original McSema.

EFACT is designed to be extensible, supporting not only binaries compiled from C/C++ but also from languages like Fortran and Rust. Additionally, it can be adapted to work with libraries like OpenSSL.

We characterized the SPEC CPU® 2017 benchmark from the perspective of EXFs, demonstrating that EXFC is an inevitable challenge. We found that ISA, optimization option, and processor architecture directly impact the number of generated EXFs. Based on our analytical results, we recommend that binary lifting researchers focusing on the ARM platform and 32-bit processor architecture should pay more attention to EXFC.

We believe our work on EFACT helps push forward static binary lifting research and we plan to continue enhancing the EXFC ability of EFACT. Currently, EFACT_MC primarily serves to evaluate the EXFC accuracy of our framework. In our experiments on cross-platform static binary translation, we observe that the detailed implementations of various data types (such as *long*) and data structures (such as *va_list*, as mentioned in Section 5.1.2) vary across platforms. These differences can lead to the crash of the translated program when being executed. We refer to these issues as the cross-platform data interpretation problem. To address this problem, we are in the process of developing a specific LLVM Pass. This Pass will perform data morphing/synchronization before/after certain external function calls within the lifted LLVM IR. Our goal is to ensure that the data types and structures are compatible with the target platform once the lifted LLVM IR is recompiled into an ELF file, thereby solving the problem as much as possible.

CRediT authorship contribution statement

Yilei Zhang: Writing – review & editing, Writing – original draft, Visualization, Software, Project administration, Data curation, Conceptualization. **Haoyu Liao:** Writing – review & editing, Validation, Software, Investigation, Data curation. **Zekun Wang:** Writing – review & editing, Software, Data curation. **Bo Huang:** Writing – review & editing, Project administration, Methodology, Conceptualization. **Jianmei Guo:** Writing – review & editing, Supervision, Project administration, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The source code of EFACT is available at https://github.com/solecnugit/EFACT.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62272167).

References

Al-Tashi, Q., Abdulkadir, S.J., Rais, H.M., Mirjalili, S., Alhussian, H., 2019. Binary optimization using hybrid grey wolf optimization for feature selection. IEEE Access 7, 39496–39508.

Altinay, A., Nash, J., Kroes, T., Rajasekaran, P., Zhou, D., Dabrowski, A., Gens, D., Na, Y., Volckaert, S., Giuffrida, C., Bos, H., Franz, M., 2020. BinRec: dynamic binary lifting and recompilation. In: Bilas, A., Magoutis, K., Markatos, E.P., Kostic, D., Seltzer, M.I. (Eds.), EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020. ACM, pp. 36:1–36:16.

Anand, K., Smithson, M., Elwazeer, K., Kotha, A., Gruen, J., Giles, N., Barua, R., 2013. A compiler-level intermediate representation based binary analysis and rewriting system. In: Hanzálek, Z., Härtig, H., Castro, M., Kaashoek, M.F. (Eds.), Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013. ACM, pp. 295–308.

Anon, 2016. Itanium-demangle. URL https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling.

Bala, V., Duesterwald, E., Banerjia, S., 2000. Dynamo: A transparent dynamic optimization system. In: Lam, M.S. (Ed.), Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000. ACM, pp. 1–12.

Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I., 2018. A survey of symbolic execution techniques. ACM Comput. Surv. 51 (3), 50:1–50:39.

Bellard, F., 2005. QEMU, a fast and portable dynamic translator. In: Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA. USENIX, pp. 41–46.

Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J., 2011. BAP: A binary analysis platform. In: Gopalakrishnan, G., Qadeer, S. (Eds.), Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. In: Lecture Notes in Computer Science, vol. 6806, Springer, pp. 463–469.

Brumley, D., Lee, J., Schwartz, E.J., Woo, M., 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In: King, S.T. (Ed.), Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013. USENIX Association, pp. 353–368.

Capstone Engine, 2023b. Capstone. URL https://www.capstone-engine.org/.

Di Federico, A., Payer, M., Agosta, G., 2017. rev.ng: A unified binary analysis framework to recover CFGs and function boundaries. In: Wu, P., Hack, S. (Eds.), Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017. ACM, pp. 131–141.

Dyninst Project, 2023c. Dyninst. URL https://www.dyninst.org/.

Eli Bendersky, 2023g. Pyelftools. URL https://github.com/eliben/pyelftools.

Embedded Microprocessor Benchmark Consortium, 2023d. EEMBC benchmarks. URL https://www.eembc.org/.

Fokin, A., Derevenetc, E., Chernov, A., Troshina, K., 2011. SmartDec: Approaching C++ decompilation. In: Pinzger, M., Poshyvanyk, D., Buckley, J. (Eds.), WCRE 2011, Limerick, Ireland, October 17-20, 2011. IEEE Computer Society, pp. 347–356.

Fu, S., Hong, D., Liu, Y., Wu, J., Hsu, W., 2019. Optimizing data permutations in structured loads/stores translation and SIMD register mapping for a cross-ISA dynamic binary translator. J. Syst. Archit. 98, 173–190.

Guan, H., Liu, B., Li, T., Liang, A., 2008. Multithreaded optimizing technique for dynamic binary translator CrossBit. In: International Conference on Computer Science and Software Engineering, CSSE 2008, Volume 5: E-Learning and Knowledge Management / Socially Informed and Instructinal Design / Learning Systems Platforms and Architectures / Modeling and Representation / Other Applications , December 12-14, 2008, Wuhan, China. IEEE Computer Society, pp. 945–952.

Guan, H., Zhu, E., Wang, H., Ma, R., Yang, Y., Wang, B., 2012. SINOF: A dynamic-static combined framework for dynamic binary translation. J. Syst. Archit. 58 (8), 305–317.

Hex-Rays, S., 2014. IDA pro: A cross-platform multi-processor disassembler and debugger.

Hong, D., Hsu, C., Yew, P., Wu, J., Hsu, W., Liu, P., Wang, C., Chung, Y., 2012. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In: Eidt, C., Holler, A.M., Srinivasan, U., Amarasinghe, S.P. (Eds.), 10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012. ACM, pp. 104–113.

Křoustek, J., Matula, P., Zemek, P., 2017. Retdec: An open-source machine-code decompiler. In: July 2018.

Lattner, C., Adve, V.S., 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. IEEE Computer Society, pp. 75–88.

Lin, Y., Gao, D., 2021. When function signature recovery meets compiler optimization. In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. IEEE, pp. 36–52.

Liu, Z., Yuan, Y., Wang, S., Bao, Y., 2022. Sok: Demystifying binary lifters through the lens of downstream applications. In: 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022. IEEE, pp. 1100–1119.

Muntean, P., Fischer, M., Tan, G., Lin, Z., Grossklags, J., Eckert, C., 2018. τCfi: Type-assisted control flow integrity for x86-64 binaries. In: Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Proceedings. Springer Verlag, pp. 423–444.

National Security Agency, 2023e. Ghidra. URL https://ghidra-sre.org/.

OpenSSL, 2023f. Openssl. URL https://github.com/openssl/openssl.

ptitSeb, 2023a. Box64. URL https://github.com/ptitSeb/box64.

Rocha, R.C.O., Sprokholt, D., Fink, M., Gouicem, R., Spink, T., Chakraborty, S., Bhatotia, P., 2022. Lasagne: a static binary translator for weak memory model architectures. In: Jhala, R., Dillig, I. (Eds.), PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022. ACM, pp. 888–902.

Rokicki, S., Rohou, E., Derrien, S., 2019. Hybrid-DBT: Hardware/software dynamic binary translation targeting VLIW. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 38 (10), 1872–1885.

Saieva, A., Kaiser, G.E., 2022. Update with care: Testing candidate bug fixes and integrating selective updates through binary rewriting. J. Syst. Softw. 191, 111381.

Sam Schlinkert, 2022a. Command-line utilities written in rust. URL https://gist.github.com/sts10/daadbc2f403bdffad1b6d33aff016c0a.

Shen, B., Chen, J., Hsu, W., Yang, W., 2012. LLBT: an LLVM-based static binary translator. In: Jerraya, A., Carloni, L.P., III, V.J.M., Rabbah, R.M. (Eds.), Proceedings of the 15th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2012, Part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012. ACM, pp. 51–60.

Standard Performance Evaluation Corporation, 2022b. SPEC CPU® 2017. URL https://www.spec.org/cpu2017/.

trailofbits, 2021. Mcsema. URL https://github.com/lifting-bits/mcsema.

van der Veen, V., Göktas, E., Contag, M., Pawoloski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., Giuffrida, C., 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In: 2016 IEEE Symposium on Security and Privacy. SP, pp. 934–953.

Verbeek, F., Bockenek, J.A., Fu, Z., Ravindran, B., 2022. Formally verified lifting of C-compiled x86-64 binaries. In: Jhala, R., Dillig, I. (Eds.), PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022. ACM, pp. 934–949.

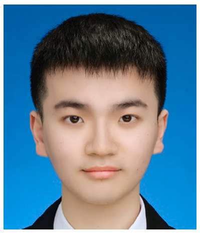
Verbeek, F., Olivier, P., Ravindran, B., 2020. Sound c code decompilation for a subset of x86-64 binaries. In: de Boer, F.S., Cerone, A. (Eds.), Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, the Netherlands, September 14-18, 2020, Proceedings. In: Lecture Notes in Computer Science, vol. 12310, Springer, pp. 247–264.

Wenzl, M., Merzdovnik, G., Ullrich, J., Weippl, E.R., 2019. From hack to elaborate technique - A survey on binary rewriting. ACM Comput. Surv. 52 (3), 49:1–49:37.

Wu, J., Dong, J., Fang, R., Zhang, W., Wang, W., Zuo, D., 2022. WDBT: non-volatile memory wear characterization and mitigation for DBT systems. J. Syst. Softw. 187, 111247.

Yadavalli, S.B., Smith, A., 2019. Raising binaries to LLVM IR with MCTOLL (WIP paper). In: Chen, J., Shrivastava, A. (Eds.), Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2019, Phoenix, AZ, USA, June 23-23, 2019. ACM, pp. 213–218.

Yilei Zhang received his bachelor's degree from Nanjing Agricultural University in Jiangsu, China, in 2019. He is currently pursuing his Ph.D. at the School of Data Science and Engineering, East China Normal University. His research interests include binary translation.



Haoyu Liao received his bachelor's degree from Chong Qing Jiao Tong University in 2022. He is pursuing a graduate degree from the School of Data Science and Engineering, East China Normal University, Shanghai, China. His research interests include performance evaluation and analysis.

Zekun Wang received his bachelor's degree from Northeastern University in 2022. He is pursuing a graduate degree from the School of Data Science and Engineering, East China Normal University, Shanghai, China. His research interests include workload sampling and analysis.

Bo Huang received his Ph.D. degree in computer science from Fudan University, China, in 2000. He is a professor with East China Normal University. With 20+ years' industry experience, his current research interests include compiler technology and data-driven system optimization.

Jianmei Guo is now a Professor at East China Normal University. His research interests include the quality assurance and performance optimization of software systems. He received his Ph.D. in Computer Science in 2011 from Shanghai Jiao Tong University. He was a Postdoctoral Fellow at the University of Waterloo, an Associate Professor at East China University of Science and Technology, and a Staff Engineer at Alibaba Group. He received two Best Paper Awards of SPLC 2016, Canadian p=AI 2017, and an ACM SIGSOFT Distinguished Paper Award of ASE 2015. He is a member of ACM, IEEE and CCF.