SoK: Demystifying Binary Lifters Through the Lens of Downstream Applications

Zhibo Liu, Yuanyuan Yuan, Shuai Wang*
The Hong Kong University of Science and Technology
{zliudc,yyuanaq,shuaiw}@cse.ust.hk

Yuyan Bao
University of Waterloo
yuyan.bao@uwaterloo.ca

***Abstract*—Binary lifters convert executables into an intermediate representation (IR) of a compiler framework. The recovered IR code is generally deemed "analysis friendly," bridging low-level code analysis with well-established compiler infrastructures. With years of development, binary lifters are becoming increasingly popular for use in various security, systems, and software (re)-engineering applications. Recent studies have also reported highly promising results that suggest binary lifters can generate LLVM IR code with correct functionality, even for complex cases.**

This paper conducts an in-depth study of binary lifters from an orthogonal and highly demanding perspective. We demystify the "expressiveness" of binary lifters, and reveal how well the lifted LLVM IR code can support critical downstream applications in security analysis scenarios. To do so, we generate two pieces of LLVM IR code by compiling C/C++ programs or by lifting the corresponding executables. We then feed these two pieces of LLVM IR code to three keystone downstream applications (pointer analysis, discriminability analysis, and decompilation) and determine whether inconsistent analysis results are generated. We study four popular static and dynamic LLVM IR lifters that were developed by the industry or academia from a total of 252,063 executables generated by various compilers and optimizations and on different architectures. Our findings show that modern binary lifters afford IR code that is highly suitable for discriminability analysis and decompilation, and suggest that such binary lifters can be applied in common similarity- or code comprehension-based security analysis (e.g., binary diffing). However, the lifted IR code appears unsuited to rigorous static analysis (e.g., pointer analysis). To obtain a more comprehensive view of the utility of binary lifters, we also compare the performance of lifter-enabled approaches with that of binary-only tools in three security tasks, i.e., sanitization, binary diffing, and C decompilation. We summarize our findings and make suggestions for the correct use and further enhancement of binary lifters. We also explored practical ways to enhance the accuracy of pointer analysis using lifted IR code, by using and augmenting `Debin`, a tool for predicting debug information.

I. Introduction

An intermediate representation (IR) denotes the language used by a compiler to represent source code for the purpose of analysis and optimization. A good-quality IR accurately and concisely represents the semantics of source code in a manner that is independent of high-level languages. IR designs may also reflect underlying hardware details or be platform-neutral.

To the best of our knowledge, SecondWrite [19], [47], was the first to advocate and provide methods for lifting executables into LLVM IR code. It enables the reuse of analysis facilities provided by the LLVM framework, thereby linking low-level code analysis with mature compiler infrastructures. New binary lifter papers are continually presented at top-tier conferences (e.g., [18], [48], [58], [113]), and industry (e.g., Microsoft [80]) has also expended considerable resources to develop binary lifters [87], [80], [68], [61].

Given the dominance of the LLVM community, most commercial binary lifters aim to translate executables into LLVM IR code. The lifting of assembly programs into LLVM IR enables a full set of LLVM compiler passes to be used to smoothly and rapidly build various security and systems applications, such as applications for vulnerability detection, malware analysis, off-the-shelf software security hardening, cross-architecture code reuse, and profiling [19], [47], [40], [113], [58]. Lifter developers have also demonstrated that lifted IR code is highly accurate and can pass most functionality tests or rigorous formal verifications [38], [66].

However, we have observed (as elaborated in Sec. III) that lifted IR code frequently exhibits visually different representations, which may stealthily undermine its utility for application to downstream tasks. Overall, there are several reverse engineering-related difficulties that arise when lifting machine code into a (platform-neutral) IR, whereas translating a high-level language into IR is generally facile. We note that the research community lacks a systematic understanding of how well the lifted IR code supports downstream applications. This has caused great confusion for normal users (e.g., [105], [21], [106], [73]), who want to use lifted IR code for various downstream applications, and is a research gap that is not fully disclosed by lifter developers [79].

This research aims to systematically determine the true capability of binary lifters, which to the best of our knowledge have yet to be explicated in real-world usage scenarios. We aim to answer the following key research question: "To what extent can lifted IR code support representative security downstream applications?" Although previous research [110] has highlighted that binary lifters' outputs may not be of satisfactory quality, recently released lifters have incorporated many advanced techniques to enhance the quality of the IR code they recover [18], [48], [113]. At present, a popular and attractive procedure involves analyzing low-level x86 binary and firmware samples by first lifting them into LLVM IR code [40], [37], [117], [39], [18]. Hence, there is a demand for studies of LLVM IR lifters in realistic settings to more clearly

*Corresponding author.

delineate the applicability of these lifters to common security analysis and instrumentation tasks.

Our study targets four modern static and dynamic binary lifters that output LLVM IR code. We set up three representative tasks, namely pointer analysis, discriminability analysis, and C decompilation, as these are the key tasks of many downstream applications that use binary lifters. We compare the analysis results obtained from the lifted IR and the compiled IR (denoted the "upper bound" quality) over three datasets comprising a total of 252,063 executables generated using various compilers and optimization settings on 64-bit x86 and ARM64 platforms. We perform extensive manual inspections and repairs of unaligned analysis results and summarize key findings and their implications. The results of discriminability analysis and decompilation are generally good (though as expected, heavy optimizations like `-O3` can impose extra challenge), and provide solid empirical support to use lifters in similarity-based analysis (e.g., patch-searching of legacy code [120], [124], [121]). Furthermore, we also compare the performance of binary-only security tools with that of lifter-enabled tools in three security tasks, sanitization [101], binary code diffing, and C decompilation, and present inspiring findings. We also study approaches to employing and augmenting `Debin` [59] to recover debug information in executables, thereby enhancing the shallow support of lifted IR code in pointer analysis. Our augmented `Debin` substantially increases the accuracy of pointer analysis supported by `RetDec` [68], a popular binary lifter. In sum, we demonstrate how binary lifters can be optimally applied to support security-related tasks via the following contributions:

- We advocate a new and important focus for the study of binary lifters. Rather than using the current approaches of testing or formally verifying the "functional correctness" of binary lifters, we explore binary lifters from an orthogonal and demanding perspective by clarifying their support for downstream applications.
- We use three downstream applications that are the cornerstone of many security analysis scenarios. To smoothly benchmark modern static and dynamic lifters, we address several engineering challenges and expend considerable manual effort. Our study is conducted in cross-compiler, cross-optimization, and cross-architecture settings. To obtain a more comprehensive overview, we also directly compare lifter-enabled solutions with binary-only solutions over three popular security tasks.
- We summarize the findings and implications of our study. We provide practical solutions to a common limitation of modern binary lifters — the ability to support rigorous static analysis — by using and augmenting recent research [59]. Our findings provide guidelines for users on how to analyze low-level binary code with LLVM infrastructures for security purposes, and also highlight further improvements that should be made by lifter developers.

We have released artifacts to support further research and enhancement of binary lifters [1].

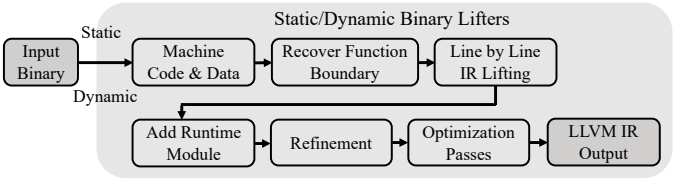


Fig. 1. The workflow of modern binary lifters. "Static" and "Dynamic" denote static disassembling and dynamic binary translation, respectively.

II. Preliminaries

Fig. 1 depicts the high-level workflow of binary lifters. To give a general review of lifting techniques, we subsume both static and dynamic lifting procedures from a very holistic perspective. Also, although LLVM IR is used as an example, the introduced process should be conceptually applicable to lifters of other IRs. We now elaborate on each step.

Reverse Engineering. A modern static binary lifter forms an "end-to-end" workflow. The input executable first goes through the reverse engineering procedures and is converted into machine code. To date, disassembling of (non-obfuscated) executable can be performed smoothly and correctly [110], [49]. Some binary lifters would even take the commercial tools (e.g., IDA-Pro [60]) as their frontend.

Dynamic lifters typically run executable within a hardware emulator (e.g., `Qemu` [25]). Executed instructions and the execution context (e.g., values of registers) are collected for lifting [48], [18]. To accelerate the discovery of new execution paths, symbolic execution engines (e.g., S^2E [35]) could be used by modern dynamic lifters [18]. Static lifters may also recover function boundary information before lifting machine instructions. To identify and recover functions, binary lifters typically outsource this task to the underlying reverse engineering infrastructures, e.g., IDA-Pro [60] or Radare2 [3].

Line by Line IR Lifting. This step denotes the key procedure to lift IR code, where each machine instruction is mapped into a sequence of IR statements. The lifted IR statements would faithfully *emulate* machine execution, including CPU register-level computations, memory updates, and other side effects. A sequence of IR statements, corresponding to one machine instruction, will be usually wrapped into a utility function. Hence, each machine instruction will be mapped to a function call in the lifted IR code to its utility function.

Runtime Module. Executing machine instructions typically updates the runtime environment, e.g., CPU registers, flags, stack, and heap. Accordingly, lifted IR code typically defines specific data structures to represent the runtime environment and assist the computation. For instance, a popular lifter, `McSema` [87], defines a runtime module of three elements `(mem, pc, state)` in its generated LLVM IR code, where `mem` represents memory and global data regions, `pc` denotes the program counter, and `state` maintains registers and CPU flags. Memory load and store are converted into querying and updating `mem` in the lifted IR code, respectively.

Refinement: From Emulation-Style IR to High-Level IR. While the *emulation* style lifting is straightforward, ex-

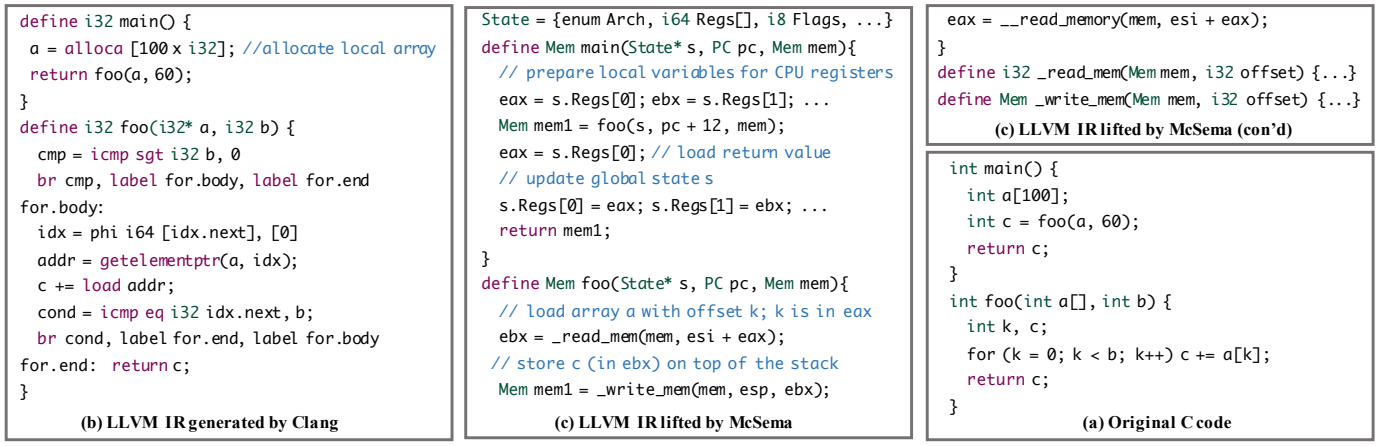


Fig. 2. A case study comparing LLVM IR lifted by `McSema` with LLVM IR compiled by `clang`. Both IR code has been extensively simplified for readability. Compiling and executing the lifted IR gives *correct* outputs, meaning that the lifted IR can smoothly pass existing functionality check research [38], [66].

isting research has pointed out considerable drawbacks in the lifted IR code. For example, while the source code (and compiled LLVM IR code) mostly uses local variables for computation, the lifted IR code likely accesses a global array (e.g., `mem` in the `McSema` case) maintained by the runtime module to mimic the usage of physical stack in machine code. This inconsistency is often referred to as the "virtual stack vs. physical stack" issue in literatures [19], [47].

To refine the lifted IR code and make it (visually) closer to the compiled IR, modern lifters can implement analysis passes or heuristics to recover local variables and variable types; successfully recovered local variables deprecate the necessity of maintaining a global array to mimic physical memory stack accesses. Precisely recovered variable types also assist static analysis (see Sec. VI-B). Nevertheless, we note that not every lifter has fully implemented such refinement procedures, as we will show in Sec. VI. See case studies in Sec. III.

Since each execution trace only partially covers the program, dynamic lifters usually merge multiple IR traces together. To this end, extra branch conditions and control transfer statements are added. PHI nodes [123] defined by LLVM IR, which stitch data flow propagations from different execution paths, are also inserted at control merge points.

Optimization. Lifters usually leverage optimization passes provided by the LLVM toolchain to optimize the lifted code. As aforementioned, each machine instruction is usually lifted into a routine function call to a sequence of IR statements, leading to very lengthy IR programs. Optimization enables to inline each function call, where considerable statements (e.g., updating a CPU flag) could be further optimized away given that they are "dead." We notice that some lifters (e.g., `RetDec` [68]) could even implement its own optimization passes to make the lifted IR code more concise [8], [6], [7].

III. Motivating Example

Fig. 2(a) presents a simple C code that sums the first 60 elements of an array. We use `clang`, the C frontend of the LLVM framework, to compile and emit LLVM IR code (see Fig. 2(b)). We use a popular static lifter, `McSema` [87], to lift the corresponding executable and present the lifted LLVM IR in Fig. 2(c). As observed, compiled IR code preserves most of the structure of the source code. In contrast, the lifted IR code represents the computation in a dramatically different way due to the following observations. 1) `McSema` uses utility functions to **emulate** the computation of each machine instruction. For example, memory loading is lifted into a callsite to `_read_mem` in Fig. 2(c). 2) `McSema` uses several globals to represent program runtime, i.e., `State` that consists of general-purpose CPU registers and CPU flags. 3) `McSema` also uses an array (i.e., `Mem` in Fig. 2(c)) to mimic the access of the x86 memory stacks.

While local variables are frequently used in compiled IR code, e.g., the variable `c` in Fig. 2(b), `McSema` uses local variables to represent registers (e.g., at the beginning of `main` in Fig. 2(c)), and updates the global state `s` before finishing function execution. Correspondingly, an array access in the original C code (e.g., `a[k]` in Fig. 2(a)) is converted into querying the global array via some utility functions, which mimics how the x86 stack is accessed in the machine code.

We emphasize that compiling and executing the lifted IR code gives *correct* outputs. Hence, existing research on either testing or formally verifying the correctness of binary lifters [66], [38], [113] would not deem the lifted IR code "specious." Nevertheless, given the dramatically different representations, one could further question how good can the lifted IR code support downstream security applications and transformations, and what are the limitations of that. To date, a thorough study regarding this point is still missing.

Emulation-Style IR (EIR) vs. High-Level IR (HIR). Fig. 2 compares compiled-IR with `McSema`-generated IR. Findings in Sec. VI will show that popular static and dynamic lifters can generate either emulation-style IR (i.e., Fig. 2(c)) or more high-level IR code that is comparable to compiled IR code in Fig. 2(b). **Refinement** in Sec. II has clarified the process of going from an emulation-style IR to high-level IR. To ease the presentation, LLVM IR compiled from source code by `clang` is referred to as compiled IR (CIR). LLVM IR yielded

by lifters following the emulation paradigms is referred to as emulation-style IR (EIR), whereas LLVM IR yielded by lifters going through further refinement as high-level IR (HIR).

Transformations Involved in Emitting EIR and HIR. With manual efforts in studying lifter codebase, we put transformations used during lifting into three categories: ① optimizations offered by the LLVM framework, ② simple optimizations developed by lifters, and ③ transformations recovering high-level program features developed by lifters. ① simplifies lifted IR code in a functionality-preserving manner. ②, e.g., removing bloated utility functions, should also be functionality-preserving if implemented correctly. However, ③ tackles a challenging task of recovering information (e.g., type) lost during compilation. According to our manual study, lifters emitting EIR (e.g., `McSema`; see Sec. V-A) leverage ① and ②. However, lifters emitting HIR typically employ ①, ②, and ③ to generate high-level code. This explains the core difficulty of producing HIR that is functional preserving; see our empirical evaluation on correctness in Sec. VI-E.

IV. Study Overview

Sec. III has clarified that we aim to study how good can the lifted IR code support downstream security applications and transformations, and what are the limitations of that. We now clarify several aspects and present our study overview.

Correctness vs. Expressiveness. Functionality correctness is a critical aspect to assess binary lifters. Functionality-preserving lifters can, in principle, support to (semi-)automatically fix a bug in legacy code or to migrate legacy code — with the end result again being a functional executable. We measure correctness of popular binary lifters and present corresponding discussions in Sec. VI-E: the results are promising and inspiring. However, we clarify that measuring correctness should *not* be the primary focus of this SoK paper due to the following reasons. 1) Existing works have launched testing and formal verification toward binary lifters [66], [38], from where we can gain an in-depth understanding on the correctness of lifters. 2) We believe that supporting static code analysis, which typically does not require to recompile and execute the lifted IR, is of equal importance with correctness. Particularly, our discriminability study in Sec. IV-B measures lifters' support of similarity analysis, which is a very actively-studied field (e.g., [51], [120], [116], [74], [124], [44], [46]). Similarly, the security community has devoted significant efforts to advancing decompilation (e.g., [59], [91], [97], [122]), and we explore lifters' support of decompilation in this work. 3) Generating flawless high-level code is a well-known challenge for reverse engineering (not just lifters). To do so, binary lifters, in principle, need to equip the "re-assembleable disassembling" scheme which is not mature [110], [108], [43], [52]. To date, decompiled C/C++ code mainly serves (human-based) process of analysis and comprehension, *not* for recompilation.

We fully agree that functionality-preserving binary lifters can extensively promote binary fixing, patching, and legacy code migration. However, these tasks, in general, can also be done on assembly code [41], [111]. In all, LLVM IR excels in promoting (static) analysis. The LLVM community has taken years of effort to develop analysis infrastructures; re-implementing those infrastructures on assembly code can take a huge effort. More importantly, assembly code does not have type or other high-level code information, which makes itself less analysis-friendly compared with LLVM IR.

Measuring IR Expressiveness. To measure correctness, existing works have used straightforward approaches involving testing or formal verification [38], [66], [113]. However, the formulation of oracles or specifications relevant to our goal — to study how good can the lifted IR support security tasks and transformations and what are the limitations of that — would be challenging, recondite, and perhaps impossible. For example, while "structuredness" of lifted IR code may be (partially) reflected by counting recovered functions, function information may not be the key to security analysis, such as buffer overflow detection. It may also be infeasible to compute the *syntactic similarity* between lifted IR and compiled IR for use as an oracle. This is due to the fact that lifters may implement different code-generation templates and tactics, thus producing syntactically different IR codes that provide comparable support for downstream tasks.

This research tackles the aforementioned challenge in a pragmatic way. We employ three downstream applications and quantify how lifted IR code supports these representative tasks. These applications profile the quality of lifted IR code from conceptually different aspects. More importantly, they are the *building blocks* of many real-world security, systems, and software re-engineering applications. This way, we ensure the inclusion of our study and the credibility of our findings.

Assessing Upper-Bound Quality Using Compiled IR. Aligned with existing works [19], [47], we deem that compiler-generated IR denotes the upper bound quality of lifted IR. This study, for the first time, empirically benchmarks "how far we are" from this perspective. Hence, instead of defining how good the lifted IR should be w.r.t. downstream tasks (which is obscure), we check whether lifted and compiled IR have *close* performance. We now introduce three representative downstream tasks.

A. Pointer Analysis

Pointer analysis establishes which pointers can point to the same variables or memory objects. Pointer analysis is the cornerstone of most data and control flow analyses, and it enables many security applications. Hence, the first part of this study examines whether pointer analysis can be launched in a fool-proof manner using lifted IR code. This reveals the *overall difficulty* of performing rigorous static analysis using lifted LLVM IR code.

We leverage the state-of-the-art LLVM pointer analysis library, `SVF` [102], [103]. `SVF` performs sparse value flow analysis to iteratively construct value flow and pointer analysis results. We use the default flow-sensitive pointer analysis [102] provided by `SVF`. Each pair of pointers is assessed by `SVF`, which determines whether they are `MustAlias`, `MayAlias`, or `NoAlias`. That is, whether they always, may, or never point to the same data region, respectively.

Variable recovery is a prerequisite of pointer analysis. SecondWrite [19], [47] has developed fully fledged variable recovery techniques, which demonstrated promising results almost a decade ago. Our preliminary study finds that some binary lifters (e.g., `RetDec` [68] and `mctoll` [80]) recover a reasonable number of variables. This indicates that benchmarking these binary lifters in terms of pointer analysis is well-timed. Furthermore, LLVM pointer-analysis facilities, including `SVF`, are highly demanding and have been widely-adopted in security research [32], [62], [64], [31], [65], [54].

B. Discriminability Analysis

The second thrust focuses on measuring the discriminability of IR code by quantifying the feasibility of determining the (dis)similarity of two pieces of IR code that implement the same or different tasks. Holistically, discriminability is the base of various *similarity-based* security applications. For instance, malware clustering and code plagiarism detection are usually conducted by analyzing the (dis)similarity between unknown software and samples of known software [118], [109], [75], [20], [112], [16], [30].

We extend a code embedding tool, `ncc` [26], that was developed in the LLVM framework. Code embedding techniques convert code into numerical vectors, such that similar codes are separated by a shorter cosine distance in numerical space than dissimilar codes. Thus, `ncc` comprehends LLVM IR code by constructing a so-called "contextual flow graph," which simultaneously incorporates IR data flow and control flow features. It then uses graph neural network (GNN)-based embedding models [125] to extract a numerical vector for each IR program. `ncc` also provides an algorithm classification model of the extracted numerical vectors, which is trained on the POJ-104 dataset [82]. The POJ-104 dataset contains 44,912 C/C++ programs that implement entry-level programming assignments for 104 different tasks (e.g., merge sort vs. two sum). Higher classification accuracy indicates that it is easier to decide the (dis)similarity of two LLVM IR programs.

C. C Decompilation

C decompilers are commonly used as the basis of many security and systems applications, including off-the-shelf software security hardening, vulnerability detection, cross-architecture code reuse, and profiling [55], [67], [107], [37], [63]. In general, C decompilers lift executables into (customized) IR, and conduct a set of analysis passes to recover high-level control structures (e.g., loops) and code patterns [28]. The `RetDec` framework [68] provides a decompiler, called `llvmir2hll`, to convert LLVM IR code into C source code. This decompiler has been shown to have comparable accuracy with commercial tools [76]. We measure whether compiled IR and lifted IR code can induce decompiled C code of similar quality.

V. Study Setup

Given a C program p, we use `clang` to compile and emit LLVM IR code p_{ir}. We then directly compile p into an executable e. We use either static or dynamic binary lifters to process e and produce a piece of lifted IR code, namely p'_{ir}. Given p_{ir} and p'_{ir} both derived from p, we feed them into downstream tasks and manually inspect potentially deviated analysis results. We aggregate the harvested information to deduce empirical findings.

We study unobfuscated ELF executable: reverse engineering is error-prone for obfuscated code. Sec. VI presents detailed evaluation when compiling e with `clang` (ver. 10.0) and no optimization. Sec. VII and Sec. VIII further explore cross-compiler, cross-optimization, and cross-architecture settings.

TABLE I
BINARY LIFTERS USED IN THE STUDY.

Tool Name	Information
`McSema` [87]	Developed by Trail of Bits, Inc.
`McSema`^{0} [87]	Disable all LLVM optimizations used by `McSema`
`mctoll` [80]	Developed by Microsoft
`RetDec` [68]	Developed by Avast
`BinRec` [18]	Published at EuroSys '20

A. Binary Lifters

Table I reports four static and dynamic binary lifters that are evaluated in our study.

`McSema`. We benchmark `McSema`, a famous static lifter that has been developed for about a decade by Trail of Bits. `McSema` performs typical emulation-style lifting, i.e., it generates emulation-style IR (EIR), as introduced in Sec. III. We report IR samples lifted by `McSema` in Fig. 2(c): its lifted IR code manifests a distinct execution mode compared with compiled IR code. Each machine instruction is emulated in LLVM IR code, and the execution context, including values of registers and memory, are passed through function callsites (see `main` and `foo` in Fig. 2(c)).

`McSema` takes decompilation and binary patching (with the end result re-emitting working binaries) as its main features. Aligned with our research motivation introduced in Sec. IV, `McSema` also champions to re-use existing LLVM-based passes and maintain one set of LLVM passes to analyze both source/binary code [87], However, our study reveals that it lacks fool-proof support for static analysis tasks. `McSema` uses a number of LLVM optimization passes to make the lifted program more succinct. To reveal *how compiler optimizations could affect the performance of downstream applications*, we configure `McSema` to disable all imposed optimization passes (referred as `McSema`^{0}). `McSema` needs a disassembler as the frontend. We equip `McSema` with IDA-Pro [60], a commercial decompiler to explore the full potential of `McSema`.

`RetDec` and `mctoll`. We also evaluate `RetDec` [68] and `mctoll` [80]. `RetDec` is a reverse engineering toolchain developed by Avast that converts executable into LLVM IR, and then decompiles the lifted IR into C code with `llvmir2hll`. Its design focus includes supporting static analysis and facilitates decompilation (e.g., by reconstructing high-level C/C++ language features) [22]. For the rest of this paper, `RetDec` refers to the lifter of this toolchain, and `llvmir2hll` refers to its decompiler. `mctoll` is an open-source project developed and maintained by Microsoft. Although not explicitly documented [81], we find that its

code generation paradigm is similar with RetDec, indicating mctoll's decent support for analysis-related tasks.

We find that RetDec and mctoll can generate LLVM IR that is visually closer to compiled LLVM IR. We consider these two lifters generate high-level IR (HIR), which is distinct with EIR lifted by McSema. While IR lifted by both lifters manifests similar visual representation, RetDec primarily recovers local variables and types, whereas mctoll emulates the computation of some CPU registers. This is likely due to challenges of recovering certain local variables. Consequently, mctoll-lifted IR code can impose higher challenge in static (data flow) analysis, as will be shown in Sec. VI-B.

BinRec. We also study a recently-released dynamic lifter BinRec (EuroSys '20 [18]). BinRec takes an executable as its input, and employs S^2E [35], a symbolic execution engine, to discover program inputs that can lead to new execution paths. S^2E runs executables within Qemu, and for each logged execution trace, a LLVM IR trace will be generated accordingly. S^2E also provides RevGen [34] to lift executables into LLVM IR code. RevGen uses IDA-Pro and McSema as its front-end [10], which overlaps with our setup of benchmarking McSema. Therefore, we do not pick RevGen for evaluation. BinRec denotes the latest dynamic lifter in this field, and *functional correctness* is an explicit design goal of BinRec (which is well demonstrated in our study of functional correctness; see Sec. VI-E). Therefore, code reuse becomes nicely feasible after recompiling lifted IR code into standalone executable.

We clarify that decompilation (and other static analysis tasks) are *not* explicit design goals of BinRec. Overall, BinRec performs emulation-style lifting; it generates standalone IR programs by merging lifted IR traces, which, in principle, eases whole-program rewriting and recompilation. BinRec places all lifted IR code into a large LLVM IR function named wrapper. This way, the original function information and the call graph are deprecated in the output of BinRec, which might cause confusion when conducting downstream tasks. This may arise, for example, during the use of binary code analyzers (e.g., BinDiff [2]), which implement call graph-based algorithms or conduct function-level analysis.

Also, while dynamic lifters usually suffer from incomplete code coverage, S^2E has shown very good support to comprehensively discover program paths [33]. Our observation shows that BinRec can achieve very good coverage for most C programs evaluated in this research, even for highly complex SPEC C programs.

We indeed spent considerable effort to explore other lifters. For instance, bin2llvm [4] is seen to produced too many pieces of broken IR code, and is no longer actively maintained. We evaluated Rev.ng [95], which performs apparently worse than other lifters and is therefore omitted. We also investigated another recently released dynamic lifter, Instrew [48], which yielded a large volume of broken outputs. Some other popular frameworks can convert binary code into (customized) low-level IR, e.g., angr [99] lifts assembly instructions into VEX IR. Our study primarily focuses on binary lifters that convert assembly programs into compiler IR; this way, rich resources (e.g., analysis passes) provided by the compiler framework could be smoothly reused in analyzing low-level code without reinventing the wheel [19], [47], [18]. In short, it might not be inaccurate to assume those four tools represent the three best static and one best dynamic lifters that convert binary code into LLVM IR by the time of writing this paper.

TABLE II
STATISTICS OF THE TEST PROGRAMS.

Total # of SVF test cases	84
Total # of alias facts	169
Total # of POJ-104 programs	44,912
Total # of SPEC INT 2006 C programs	9

TABLE III
CHANGES WE MADE TO BINARY LIFTERS AND RELEVANT TOOLS.

Tool	Changes
McSema	Remove compiler and linker inserted functions
McSema	Create McSema^0 by disabling all optimizations
BinRec	Several enhancements and bug fixes
mctoll	Support 20 new external function calls
ncc	Completely rewritten in PyTorch [89]
llvmir2hll	Add ten unsupported LLVM IR instructions

B. Downstream Task Setup and Test Case Selection

Table II summarizes the statistics of our test cases. We now elaborate on the setup of each downstream task as follows.

Pointer Analysis. We use all 24 flow-sensitive test cases shipped by SVF. For each test program, some pointer pairs are annotated with MustAlias, MayAlias, or NoAlias. They are the ground truth of pointer analysis. We study whether lifted and compiled IR code can support SVF to generate consistent and correct pointer analysis results w.r.t. the ground truth. Note that these 24 test cases, though encoding different flow-sensitive pointer analysis challenges, have relatively simple control structures. To increase the difficulty, we also perform equivalence modulo inputs (EMI)-based mutation [69] to generate extra 60 programs with more complex program structures. EMI-based mutation has been widely used to mutate C code and test compilers [69], [104], [70]. We manually confirm and adjust alias predicates in case they were changed by EMI mutations (e.g., after EMI mutation, a pair of "must not alias" pointers could become "may alias"). 84 test programs contain in total 169 pointer alias facts to check.

Discriminability Analysis. We use the default setting in the ncc paper [26] to split POJ-104 programs into training, validation, and testing sets. Each dataset will be either compiled or lifted into LLVM IR programs. That is, we will create five training, five validation, and five testing datasets. We then train one ncc LLVM IR embedding model and one classification model of POJ-104 using a training/validation dataset. The trained models are tested using the corresponding test dataset. The model training is completed at 50 epoch. We use the default model hyper parameter settings specified by ncc.

C Decompilation. We feed lifted and compiled IR code into llvmir2hll, and measure decompilation quality. We collect all C programs from the SPEC INT 2006 test suite (in total

TABLE IV
POINTER ANALYSIS EVALUATION RESULTS. SVFG DENOTES SPARSE VALUE FLOW GRAPH [102], WHOSE SIZE INDICATES THE *complexity* OF PROGRAM STATIC ANALYSIS. WE REPORT THE AVERAGE DATA FOR POINTERS, OBJECTS, SVFG GRAPHS, AND PROCESSING TIME EVALUATION.

Tool	`MustAlias` Accuracy	`MayAlias` Accuracy	`NoAlias` Accuracy	#Pointers	#Objects	#SVFG Nodes	#SVFG Edges	Processing Time (CPU Seconds)
`Clang`	100.0%	100.0%	100.0%	345.9	68.6	164.7	116.4	2.5
`RetDec`	0.0%	0.0%	17.8%	73.0	14.2	25.2	8.8	3.9
`mctoll`	0.0%	0.0%	0.0%	934.8	52.9	168.8	102.0	5.7
`McSema`	0.0%	0.0%	0.0%	3393.1	141.2	953.5	817.6	19.8
`McSema`^{0}	0.0%	0.0%	0.0%	96646.2	2682.5	26686.1	20802.8	463.5
`BinRec`	0.0%	0.0%	0.0%	879.4	41.9	166.7	127.7	5.7

TABLE V
TOTAL TEST CASES AND FAILED LIFTING CASES.

Dataset	`SVF`	POJ-104	SPEC
Total test cases	84	44,912	9
`RetDec`	0	2	0
`mctoll`	14	23,351	9
`McSema`	0	85	0
`McSema`^{0}	0	72	0
`BinRec`	2	9,668	0

nine programs) in this study. We exclude 3 C++ programs in the SPEC INT 2006 dataset, since decompiling C++ code is highly challenging, and is not supported by `llvmir2hll` which only converts LLVM IR code into C code.

C. Changes Made on IR Lifters and Downstream Applications

Table III reports augmentations we made on the lifters and downstream applications. Overall, we spent **considerable manual efforts**, which are documented at [11] and released at [1]. Particularly, while `llvmir2hll` is part of the `RetDec` framework, it is mature enough to decompile IR generated by both `RetDec` and `clang`. As for IR generated by other lifters, a few statements are not supported, given that some lifters seem to use LLVM IR statements that are not commonly-picked by `clang`. We patched `llvmir2hll` to support all encountered statements. With these patches, `llvmir2hll` can decompile IR lifted by other lifters.

VI. FINDINGS

This section presents our findings when lifting executables compiled with `clang` and no optimizations. We present cross-platform, cross-compiler, and cross-optimization evaluation in Sec. VII and Sec. VIII. We compare lifter-enabled solutions with binary-only tools in Sec. IX.

A. Binary Lifting Results

We compile all test cases shown in Table II into 64-bit x86 executables as the lifter inputs. We report the statistics of lifting failures in Table V. When lifting `SVF` programs, `mctoll` throws exceptions for seven cases, and also generates seven lifted LLVM IR code that are broken. When processing POJ-104 test cases, `RetDec` throws two exceptions. `mctoll` cannot process C++ programs, leading to 23,351 failures to lift POJ-104 programs, and also fails to lift all nine SPEC programs (see our artifacts [1] for error logs).

Static lifting of `SVF` and POJ-104 programs can be finished within a few seconds, whereas lifting SPEC programs can take several minutes to a few hours. `BinRec` is slower than others, as it uses symbolic execution to explore program paths. However, as `SVF` test programs are small, there are only two programs that cannot be processed by `BinRec`. When performing symbolic execution of the POJ-104 programs, we set the timeout as 10 minutes; 9,668 (21.6% of) programs cannot be lifted within this threshold. The function coverage is 93% for successfully-lifted programs. We also emphasize that these failures can be circumvented when configuring S^2E with concrete inputs, or replacing with better symbolic execution engines. This is *not* the fault of `BinRec`. As for SPEC programs, we conduct symbolic execution for each case for 50 hours, as these are much larger than other programs. We obtain an average function coverage of 23%. Such coverage should not impede the measured quality of decompiled C code, as will be explained in Sec. VI-D. We also assess cross-compiler, cross-optimization, and cross-architecture settings, and the results of lifting executables under these settings are given in Sec. VII and Sec. VIII, respectively.

B. Pointer Analysis

Table IV reports the evaluation results. As mentioned above, `SVF` test cases are annotated with the pointer alias ground truth. We count analysis results that are not equal to the ground truth as failures of lifters. For results that are equal to the ground truth, we manually inspect the details and confirm those are not false positives. Note that when lifters generate IR code with broken annotations (see **Manual Study** below; "annotations" are special function callsites using two pointers), `SVF` might also report `NoAlias`; these are false positives.

While compiled IR supports fool-proof analysis, Table IV shows that lifted IR has trivial support. No `MayAlias` or `MustAlias` relations can be correctly recognized (see discussion below). `SVF` also reports #pointers and #memory objects identified from each program. Each pointer may point to null, one, or several objects. We also report #nodes and #edges on the sparse value flow graph (SVFG) [102]. `SVF` performs pointer analysis by first constructing the SVFG, denoting the data dependency among program variables. These four criteria indicate the complexity of conducting static analysis (*not* merely pointer analysis) since the more complex program data dependency is, the higher the complexity that static analysis could encounter. We also report average processing time of `SVF`, which is consistent with the graph complexity.

Manual Study. `SVF` test cases annotate alias by placing a pair of pointers in a special function call, as follows:

```
MAYALIAS(p,q); //p and q should be ''may alias''
```

TABLE VI
INSPECTING 24 SVF IR PROGRAMS LIFTED BY RETDEC.

Correct	Inconsistent & Not Fixable	Inconsistent & Fixable
1	7	16

When the analysis pass of SVF reaches a callsite of MayAlias during its traverse of the LLVM IR control flow graph, it checks whether the two parameters of this callsite point to the same object. NoAlias and MustAlias are annotated similarly. However, McSema completely changes the callsite interfaces. As mentioned in Sec. V-A, functions in its lifted IR are in the following form:

```
define @MAYALIAS(State %s, PC %pc, Mem* %memory)
```

from which SVF cannot extract two target pointers for checking. Similarly, as introduced in Sec. V-A, function information is trimmed from the outputs of BinRec. Therefore, function callsites like MayAlias are absent in the lifted IR code.

Sec. V-A introduces that RetDec emits succinct and visually superior IR code, while mctoll sometimes emulates computation of CPU registers, which may be due to its struggling to recover certain variables. mctoll thus produces much more obscure IR code than RetDec, as shown in the "#Pointers" column of Table IV.

RetDec marginally outperforms others by generating several true positives. We also notice that the average SVFG graph of RetDec-generated IR is smaller than that of compiled IR. This is due to the heavy optimizations that are applied by RetDec (see discussions in Sec. VI-C). To determine which aspect of the lifted IR code can be enhanced for pointer analysis, we manually check and fix inconsistencies in the RetDec-lifted IR to correct the analysis results. We iterate all supplied SVF test cases (24 programs), but exclude the 60 EMI-based mutations, as they are too complex to be fixed manually. The manual confirmation results are presented in Table VI; these show that all of the alias facts of only one program are correctly analyzed, and hence do not need to be fixed. Seven programs are too challenging to fix, as SVF's flow-sensitive analysis conservatively analyzes arrays, whereas RetDec does not recover arrays in the lifted IR at all. We manually fix the remaining 16 cases to correct the pointer analysis results. The results are given in Table VII.

RetDec exhibits reasonable accuracy in recovering variables (except those from arrays, as revealed in Table VI). However, inaccurate type-inference impedes pointer analysis; 213 cases are caused by incorrectly treating a local variable of pointer type (e.g., i8*) as an integer (e.g., i32). We also find many function prototype-recovery errors (e.g., a parameter of i32* type is recovered as i32). We also add a few new statements and adjust existing statements (66 in total).

Findings. Our observation shows that modern (static) binary lifters, particularly, RetDec and mctoll, can recover variables reasonably well (see Table VII). They also strive to infer types and function prototypes, although the accuracy is low. As a result, the induced IR code can hardly support rigorous static analysis. We summarize two major obstacles that could be addressed: 1) *Type recovery*, particularly, at least distinguishing variables of pointer and non-pointer types. Recovering composite data types like C array and struct is not yet supported and could be challenging [100]. 2) *Function prototype recovery*, particularly, generating more "high-level" function prototype aligned with compiled IR code. Function prototypes generated by McSema are highly inconsistent with clang, RetDec, and mctoll generated IR code.

TABLE VII
MANUALLY FIXING 16 SVF LLVM IR PROGRAMS LIFTED BY RETDEC TO CORRECT POINTER ANALYSIS RESULTS.

Programs	Variable Type	Function Prototype	Added/Tweaked Statements	Variable Recovery
branch_1.c	5	2	2	
branch_2.c	6	4	0	
branch_3.c	20	4	4	
strong_update.c	1			
global_1.c	25	4		2
global_2.c	28	4	2	
global_3.c	16	2	3	
global_5.c	20	2	2	
simple_1.c	4	4	2	
simple_2.c	6	4	3	
simple_3.c	4	4	2	
struct_1.c	6	4	2	
struct_2.c	6	4	2	
pcycle1.c	6	4	1	
pcycle2.c	41	2	36	
test_su.c	19	2	5	
Total	213	50	66	2

We emphasize that the aforementioned issues are **common limitations** of binary lifters. Some engineering mismatches of McSema and BinRec hinder the pointer analysis passes of SVF. More importantly, our manual inspection shows that McSema and BinRec struggle to recover the variables of pointer types. Thus, the fixing of shallow "engineering mismatches" in McSema and BinRec cannot substantially enhance the pointer analysis accuracy. Overall, RetDec marginally outperforms the other binary lifters for this task by performing well in recovering variables and recovering a small number of variable types. It also preserves the callsite prototype (e.g., the callsites of the MayAlias utility function). Nevertheless, even for this "best" tool, we find that it implements rudimentary inference passes [8], [6], [7]. Its inference modules therefore require principled improvement, rather than only minor engineering-level modifications.

Possible Enhancements. Our further manual study on the source code of binary lifters shows that inconsistencies exposed in this evaluation (e.g., type recovery failures) are **not** primarily due to bugs. Rather, lifters have not fully implemented research products in this field (to date, relevant passes are generally rudimentary [8], [6], [7]). Existing research has proposed static analysis approaches for variable recovery and type inference, such as Value Set Analysis (VSA) [23], [24], [72]. Function prototype recovery can be conducted following very similar principles or with AI techniques [36], [19]. Also, we note that binary lifters such as McSema employ a commercial tool, IDA-Pro [60], as its reverse engineering frontend. In addition to function boundary information currently being acquired from IDA-Pro, we suggest McSema to leverage function prototypes already inferred by IDA-Pro. Currently, IDA-Pro is not fully used by McSema, which might be due to

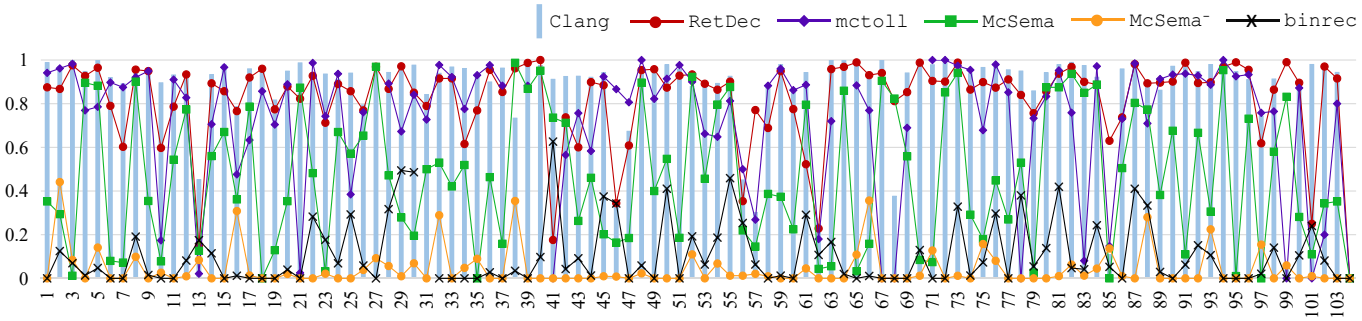


Fig. 3. Classification analysis results. We report accuracy of each tool across all 104 classes of the POJ-104 dataset.

"open-source" concerns (see a note from the lead developer of McSema [56]). Overall, we advocate lifter developers to adopt research products or explore the full potential of its employed third-party tools. It can thus ease the adoption of analysis facilities provided by the LLVM ecosystem. See Appx. E for further exploration of promoting pointer analysis.

TABLE VIII
AVERAGE CLASSIFICATION ACCURACY. HIGHER ACCURACY INDICATES BETTER SUPPORT OF VARIOUS SIMILARITY-BASED APPLICATIONS.

Clang	87.3%	McSema	46.0%
McSema^0	4.3%	RetDec	81.9%
mctoll	77.4%	BinRec	11.6%

C. Discriminability Analysis

Discriminability analysis is evaluated to classify POJ-104 programs. Fig. 3 reports the classification accuracy across 104 classes, with the average number given in Table VIII. As aforementioned, mctoll does not lift executable compiled from C++ programs, and BinRec failed over 9K cases. Therefore, their accuracy of some classes (e.g., 27) are omitted.

McSema has an average accuracy of 46.0%, while McSema^0, with no optimization applied, drops to barely functional. This inconsistency illustrates the importance of using code optimization. Fig. 3 shows that RetDec is even better than clang for certain classes (e.g., class 57). This is reasonable: when compiling POJ-104 programs into IR using clang, we disable any optimization (-O0). In contrast, RetDec optimizes lifted LLVM IR code using its own optimization and LLVM optimization passes. Our manual study shows that RetDec applies 38 optimization passes when lifting an executable file, where 26 are from the LLVM framework, and 12 are implemented by its own.

Sec. IV-B has introduced that ncc extracts contextual flow graph [26], denoting how data flow is propagated and how control structures can influence data propagations. Fig. 2 has motivated this study by showing that McSema maintains local variables in an emulation-style paradigm as follows:

```
define @main(State %s, i64 %pc, Mem %memory) {
  eax = %s.Regs[0]; ebx = %s.Regs[1]; ...
  ... // operations among vars. and %memory
```

where some LLVM identifiers are initialized at the beginning of every function and used to mimic CPU registers. A global memory array, representing memory stack, is shared and accessed by all functions. Such a paradigm, while faithfully emulating the machine code computation, exhibits distinct data access and usage patterns, undermining discriminability analysis. In other words, POJ-104 programs of different categories become undesirably indistinguishable since their data flow patterns are too similar.

BinRec shows relatively lower accuracy for this evaluation due to similar reasons. Although the average function coverage rate of BinRec is 93% when processing POJ-104 programs, the lifted code is put into a unified wrapper function, breaking the original call graph. Our manual study on the wrapper functions shows that it is generally difficult for human to identify the similarity of two wrapper functions implementing different tasks. Typically, a few thousand IR statements, corresponding to several logged execution traces, are merged within one wrapper function. Local variables are named aligned with CPU registers (R_ECX, R_ESP, etc.), mimicking the machine computation occurred on the logged execution traces.

For certain classes, both compiled and lifted IR have low accuracy. For instance, many IR programs of class 85 are labeled as class 100. After inspecting the source code, we find that programs of class 85 and class 100 have similar data structures and access patterns, i.e., a matrix is defined and each cell in the matrix is updated using its adjacent cells.

Findings & Lessons. Lifters like RetDec and mctoll show encouraging support for discriminability analysis, whose performance is *comparable* to compiled IR code (i.e., the "upper bound"). Sec. IX and Appx. B further compare lifter-enabled approaches and binary-only tools over binary diffing tasks, which induce consistent findings. In sum, our findings reveal promising progress and continuous effort made by the lifter developers and the community. It might not be inaccurate to assume that these findings increase the confidence for security researchers to use binary lifters, particularly RetDec and mctoll, in similarity analysis and binary diffing-related tasks, e.g., code patch search [120], [124], [121].

Possible Enhancements. Our manual study shows that *program optimization* is critical in generating concise and discriminable IR code. We note that Table VIII has shown a significant improvement comparing McSema with McSema^0. That is, by enabling LLVM optimization passes (eliminating deadcode, inlining routine functions, etc.), even the emulation-

style lifting can exhibit non-trivial support of discriminability analysis. Again, optimization passes customized by RetDec (e.g., [8], [6], [7]) can be referred by developers of other lifters as the starting point for enhancement.

Our study advocates for *high-level lifting* (see research motivation in Sec. III) instead of *emulation-style lifting* which mimics the physical stack with a global array. RetDec and mctoll, when reasonably meeting this criterion, have much better performance that is close to compiled IR code. RetDec has certain disassembling flaws (failed to recover certain code components; also noted in Sec. VI-E), which likely explains ways to further enhance its accuracy (i.e., eventually reaching 87.3%). Regarding local variable recovery, the SecondWrite paper [19] has proposed symbolic execution-based approaches and demonstrates the scalability on complex SPEC programs. Appx. E further assesses a recent AI-based approach [59] to predicting debug information from stripped binary code, which could boost local variable recovery as well.

Threat To Validity: Potential Bias of DNN-Based Discriminability Analysis. Many recent works on binary code similarity analysis employ neural networks and its enabled representation learning (e.g., [44], [46]). However, to our knowledge, ncc is the *only* tool that presents an end-to-end solution to generate embeddings of LLVM IR code which can smoothly support discriminability (similarity) analysis. It could be difficult to estimate the effort to migrate other code embedding tools (e.g., [44], [17]) to the LLVM ecosystem.

We clarify that ncc *not* merely builds on syntactic information. It is trained over the contextual flow graphs of LLVM IR [26]. *Semantics*-level features, including both data- and control-flow, are leveraged to construct such contextual flow graphs. We admit that it is still an open problem for DNN models to precisely reason about code semantics (e.g., analyzing code inlining or eliminating dead code). However, more rigorous semantics-based approaches (e.g., using symbolic execution and constraint solving for code similarity) generally suffer from scalability issues and are mostly limited to analyze execution traces or basic blocks (e.g., [53], [78]). To the best of our knowledge, ncc represents the latest *out-of-the-box* solution offered by the LLVM community to deliver well-performing and scalable analysis of code similarity.

Also, explaining DNN predication is still challenging (especially for human), which could depend on subtle embedding representations and model design. We leave it as a future work to leverage recent advances in explainable AI (XAI) [57] to further interpret the predictions of ncc.

D. Decompilation Analysis

With the prosperous development of automated analysis and retrofitting utilities on compiled IR code, C code decompilation retains its importance by supporting (layman) users for code comprehension and reuse. Software decompilation, in general, is to generate high-level source code for the (human-based) process of analysis and comprehension, not for (automated) recompilation. Given that structured code is shown as easier for human to understand [42], [83]. decompiler outputs with fewer "unstructured" control flow statements, i.e., goto statements, deem better quality [28]. We measure *structuredness* of decompiled C code by reusing the metrics in [28] to count #goto. We also measure the average LOC. Lengthy C code implies relatively low readability for even experts. Since McSema and McSema^0 contain many utility functions to emulate machine instructions, we count average LOC per *user-defined* function instead of the entire decompiled code. We also clarify that these metrics are *not* applicable to assess quality of lifted IR, since #goto and LOC likely won't affect automated static analysis and comprehension on LLVM IR.

TABLE IX
AVERAGE STRUCTUREDNESS AND LOC PER USER-DEFINED FUNCTION OF SPEC PROGRAMS.

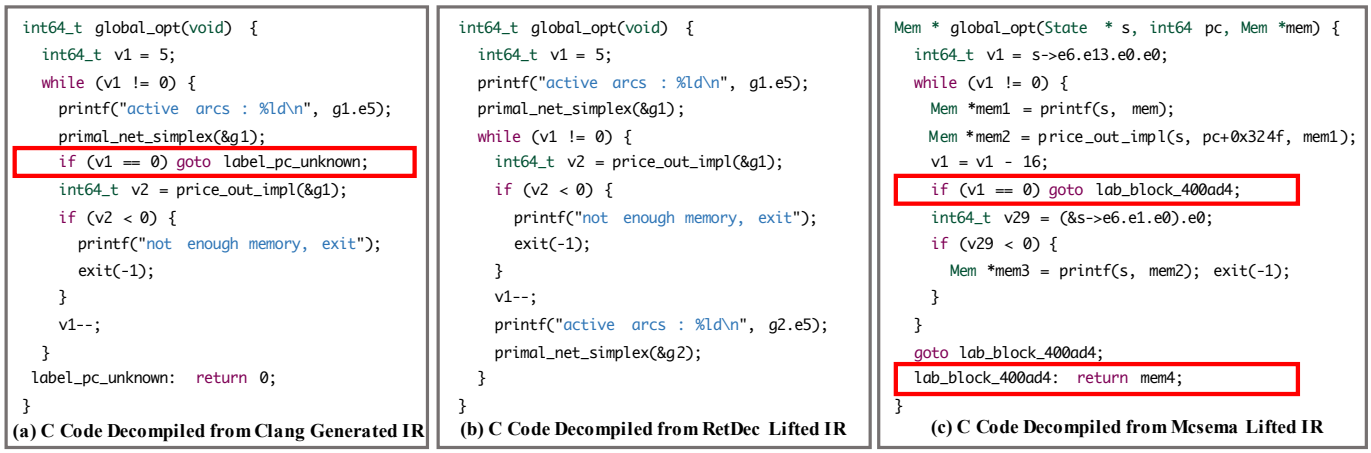
	clang	McSema	McSema^0	RetDec	BinRec
#goto	3188.4	6026.5	6926.8	1709.6	6344.1 (27894.4)
LOC	44.3	247.6	325.5	168.7	70947.1

Table IX reports the structuredness evaluation results. As discussed in Sec. VI-B, mctoll failed lifting all SPEC programs. Therefore, mctoll is omitted in Table IX. In addition, llvmir2hll cannot finish decompiling LLVM IR code of 403.gcc and 400.perlbench lifted by BinRec within two weeks. Decompiling 403.gcc IR code lifted by RetDec has similar timeout issue. 403.gcc and 400.perlbench are two largest SPEC C programs. Therefore, we exclude these two programs when reporting Table IX.

Among all the lifted C code, RetDec shows even fewer goto statements compared with clang (see **Findings** for discussions). In contrast, McSema and McSema^0 retained almost double amount of goto statements. As aforementioned, BinRec covers on average 23.0% functions for each SPEC program, inducing on average 6344.1 goto statements per program. As a result, we estimate that BinRec would generate about 27894.4 goto statements if full coverage was achieved. We also obtained consistent results for the LOC per function evaluation: compiled IR exhibits best performance for this criterion, followed by RetDec and McSema. BinRec maintains the entire lifted IR code into a huge "wrapper" function, thus manifesting relatively higher LOC.

Fig. 4 presents a case study by comparing decompiled C code of function global_opt in 429.mcf. A goto statement appears within the loop in Fig. 5(a), corresponding to a control transfer in the IR code. Accordingly, our manual study shows that RetDec has already optimized away that seemingly useless control transfer. This indicates the importance of code optimization. McSema (and McSema^0) shows likely sloppy code patterns, where a (useless) goto is placed right before its targeted label. This code pattern might partially explain that McSema has almost double remaining #goto.

While such special goto patterns might be easy to pinpoint and elide, Fig. 4 has also revealed the low readability of C code decompiled from McSema generated IR. C code in Fig. 4 has been largely simplified: the global_opt function decompiled from McSema generated IR has 298 LOC, and the same function decompiled from McSema^0 generated IR has 234 LOC; both functions are verbose and hard to read.



```
int64_t global_opt(void) {
  int64_t v1 = 5;
  while (v1 != 0) {
    printf("active arcs : %ld\n", g1.e5);
    primal_net_simplex(&g1);
    if (v1 == 0) goto label_pc_unknown;
    int64_t v2 = price_out_impl(&g1);
    if (v2 < 0) {
      printf("not enough memory, exit");
      exit(-1);
    }
    v1--;
  }
 label_pc_unknown: return 0;
}
```

(a) C Code Decompiled from Clang Generated IR

```
int64_t global_opt(void) {
  int64_t v1 = 5;
  printf("active arcs : %ld\n", g1.e5);
  primal_net_simplex(&g1);
  while (v1 != 0) {
    int64_t v2 = price_out_impl(&g1);
    if (v2 < 0) {
      printf("not enough memory, exit");
      exit(-1);
    }
    v1--;
    printf("active arcs : %ld\n", g2.e5);
    primal_net_simplex(&g2);
  }
}
```

(b) C Code Decompiled from RetDec Lifted IR

```
Mem * global_opt(State * s, int64 pc, Mem *mem) {
  int64_t v1 = s->e6.e13.e0.e0;
  while (v1 != 0) {
    Mem *mem1 = printf(s, mem);
    Mem *mem2 = price_out_impl(s, pc+0x324f, mem1);
    v1 = v1 - 16;
    if (v1 == 0) goto lab_block_400ad4;
    int64_t v29 = (&s->e6.e1.e0).e0;
    if (v29 < 0) {
      Mem *mem3 = printf(s, mem2); exit(-1);
    }
  }
  goto lab_block_400ad4;
  lab_block_400ad4: return mem4;
}
```

(c) C Code Decompiled from Mcsema Lifted IR

Fig. 4. Decompilation case study of `429.mcf`. The decompiled code, in particular the `McSema` case, is extensively simplified for readability. Indeed, `global_opt` decompiled from `clang` and `RetDec` generated IR has 32 and 45 LOC, respectively. In contrast, `global_opt` decompiled from `McSema` and `McSema`^{0} generated IR are obscure and verbose (298 and 234 LOC).

Findings. To date, decompiled C code is mostly used by experts for code comprehension. Although we have extensively simplified the sample code in Fig. 4, it should be accurate to assume that any users with elementary C programming background can spot the (syntactic) difference between Fig. 4(a) and Fig. 4(c). In contrast, Fig. 4(a) and Fig. 4(b) are closely correlated. In addition to good support for discriminability analysis, this section has reported findings that compact IR code recovered by `RetDec` largely improves the readability and structuredness of its decompiled C code. Soon we will show that decompilation evaluation over cross-compiler, cross-optimization, and cross-architecture settings also report consistently encouraging findings (see Sec. VII and Sec. VIII).

In addition to (commercial) decompilers which can generate relatively more structured C code (see Sec. IX), show that there are free and highly extensible lifter-based solutions with decent performance. Consistent with findings in discriminability analysis, we interpret that: 1) *code optimization* plays a critical role in supporting decompilation and generating C code of better quality, and 2) *recovering local variables* helps to eliminate the emulation-style lifting, enabling the generation of succinct C code with higher readability.

Possible Enhancements. Overall, our findings show that the *high-level lifting* promotes both discriminability analysis and C decompilation. Hence, we envision that **Possible Enhancements** elaborated in Sec. VI-C can also be leveraged to enhance support of decompilation. Moreover, Fig. 4 has shed lights on conducting user survey as a future direction to summarize more findings on the (un)structuredness of C code decompiled from lifted IR code. Those findings can provide practical feedback to fine tune lifters.

E. Functionality Correctness of Lifted IR Programs

Existing research has laid a solid foundation on testing or formally verifying binary lifters [66], [38], [37]. Hence, measuring the functionality correctness is *not* our primary focus. Nevertheless, we still recompile the lifted IR code and check their execution results to compare with other metrics.

`SVF` programs are relatively simple with no execution outputs. Hence, we omit `SVF` programs in this evaluation. We select one POJ-104 program from each class (in total of 104 programs). Since no documents are shipped by POJ-104, we manually write test inputs by reading the source code. We successfully wrote non-trivial inputs and acquired the corresponding outputs for 86 programs. These input/output pairs are used to check the correctness of the lifted IR code. For nine SPEC programs, we use their shipped scripts for testing. The results are reported as follows:

	`McSema`	`McSema`^{0}	`RetDec`	`mctoll`	`BinRec`
POJ-104	94.2%	94.2%	20.3%	22.1%	100.0%
SPEC	44.4%	44.4%	0	0	100.0%

Most IR programs lifted by `McSema` (and `McSema`^{0}) can pass the test cases. `RetDec` and `mctoll` show lower success rates; we find that their successful cases are largely overlapped (i.e., 16 simple POJ-104 programs), indicating that they show reasonable correctness for relatively simple cases. A few IR programs lifted by `RetDec` have reverse engineering failures (annotated with "undefined_function" in its outputs), impeding recompilation. We use the `test`, `ref`, and `train` input sets of SPEC to check the correctness. Four ($\frac{4}{9} \approx 44.4\%$) SPEC programs lifted by `McSema` and `McSema`^{0} can pass the functionality check. Given that SPEC test cases have been evaluated in the `BinRec` paper [18], we run POJ-104 cases. As suggested by the `BinRec` authors, we configure S^{2}E with concrete inputs to lift POJ-104 cases. The evaluation is highly encouraging: all test cases can be correctly lifted into IR and then recompiled into another piece of executable.

Findings. `McSema` shows highly promising results for generating functionality-preserving IR code. In Sec. IX and Appx. A, we further show that by generating functional code, `McSema` exhibits good support for sanitization [101] (as sanitized code needs to be *executed* instead of *statically analyzed*). Nonetheless, as reported in this section, `McSema`'s output is of low expressiveness. It does not suffice supporting security *analysis* as good as compiled IR code, nor is it comparable to that of other lifters, such as `RetDec` and

TABLE X
EVALUATION RESULTS FOR EXECUTABLES ON THE ARM64 PLATFORM.

Lifter	Pointer Analysis			Discriminability	C Decompilation		Functionality	
	MustAlias	MayAlias	NoAlias	Classification Accuracy	#goto	LOC	POJ-104	SPEC
RetDec	0.0%	0.0%	16.9%	74.8%	1364.6	97.5	18.2%	0.0%
McSema	0.0%	0.0%	0.0%	53.5%	12184.2	252.6	0.0%	0.0%

mctoll. Overall, we summarize that modern binary lifters are seen to have *distinct design focus*, inducing different levels of support for downstream tasks. This aspect, despite its importance, is generally ignored by the community and could cause great confusions in various security usage scenarios.

Lessons. It might be accurate to summarize inspiring *meta-lessons* from the functionality evaluation: *emulation lifting* adopted by McSema and BinRec generates LLVM IR code that can be smoothly recompiled and executed. Its output, however, is likely to be low-level that it is not analysis-friendly. Hence, users aiming to recompile the lifted LLVM IR code can opt for emulation-style lifters for better functionality correctness guarantee. In contrast, *high-level lifting* with more aggressive optimizations could be used in case users opt for more "expressive" LLVM IR code to support analysis. Also, LLVM IR is platform independent; hence, emulation lifting can support *cross-platform* profiling and recompilation [18], while existing functionality-preserving disassembling (e.g., [110], [108], [49], [43]) can only support assembly code reuse on the *same platform*.

VII. CROSS-PLATFORM EVALUATION

Existing research has been using (customized) binary lifters to convert firmware samples into LLVM IR for security analysis (e.g., [37], [40]). In this section, we further explore lifting binary code compiled on the ARM architecture. RetDec and McSema support lifting binary code compiled on the 64-bit ARM platform. Appx. D reports the binary lifting results; only McSema made a few (less than 1%) lifting failures.

We report evaluation results in Table X. We interpret that de facto lifters show *comparable* support on 64-bit x86 and ARM64 platforms. In particular, discriminability analysis reports mostly consistent results compared with Table VIII. Both RetDec and McSema show low support for pointer analysis. This is intuitive: key findings that impedes pointer analysis of RetDec and McSema in Sec. VI-B are *platform independent*.

Table X shows that C code decompiled from McSema-generated IR code contains more goto. Recall llvmir2hll cannot decompile the LLVM IR code of 403.gcc in our decompilation evaluation on x86 platforms (Sec. VI-D). We report that the LLVM IR code of 403.gcc can be successfully decompiled, and therefore, #goto in the 403.gcc case is taken into account when computing the average results in Table X. This case contributes over 70K goto statements, thus largely increasing the average results.

McSema also shows surprisingly low support for functionality correctness. Our manual study shows that McSema generates an incorrect wrapper for the main function. That is, the LLVM IR program entry point of all lifted cases are malfunctional due to bugs. We show this erroneous code pattern at [11]. We have reported this issue to the McSema developers.

TABLE XI
EVALUATION FOR CROSS-COMPILER AND CROSS-OPTIMIZATION SETTINGS. "CLANG -O0" HAVE BEEN REPORTED IN SEC. VI.

	Lifter	Discriminability Avg. Accuracy	C Decompilation	
			#goto	LOC
gcc -O0	RetDec	78.8%	1661.3	142.3
	mctoll	55.2%	NA	NA
	McSema	15.4%	15479.3	282.7
	BinRec	22.3%	6661.4(12978)	245824.2(478924.3)
gcc -O3	RetDec	78.3%	3766.4	374.3
	mctoll	41.2%	NA	NA
	McSema	10.8%	8023.2	595.1
	BinRec	11.0%	1820(20078.2)	61677(680419.3)
clang -O0	RetDec	81.9%	1709.6	168.7
	mctoll	77.4%	NA	NA
	McSema	46.0%	6026.5	247.6
	BinRec	11.6%	6344.1(27894.4)	70947.1(311947.6)
clang -O3	RetDec	79.6%	8971.5	276.6
	mctoll	54.6%	NA	NA
	McSema	30.2%	9367.0	554.6
	BinRec	4.7%	1635(3448.7)	36256.7(76476.1)
clang -O0	NA	87.3%	3188.4	44.3

VIII. CROSS-COMPILER & OPTIMIZATION EVALUATION

Sec. V has mentioned that we use clang without any optimizations to launch experiments in Sec. VI. Although the lifted IR exhibits poor support for pointer analysis, we find that lifters can generate IR code from non-optimized executables that has quality *comparable* to that of compiled IR used for discriminability analysis and C decompilation. This section further assesses the generalizability of our findings in terms of cross-compiler and cross-optimization settings. We use gcc (ver. 7.5.0) and we also use full compiler optimizations (-O3) for the evaluation. McSema[0] gives notably worse results than McSema, and is thus not evaluated. We report the binary lifting results in Appx. D, and these are generally consistent with the lifting results reported in Table V.

We summarize the results for different settings in Table XI; the last row represents clang-generated LLVM IR. For the decompilation evaluation of BinRec, a dynamic tool, data in the parentheses denotes the estimation of induced #goto and LOC if full coverage was achieved. mctoll fails to lift all SPEC programs, and its results are "NA". For discriminability analysis, RetDec manifests high robustness toward different compilation/optimization settings. In contrast, mctoll and McSema show worse performance when applying full optimizations or using the gcc compiler. mctoll gives relatively poor support for gcc compiled executable; specifically, our manual study shows that lifting some gcc compiled executables silently generates broken IR fragments, without giving any warning. As admitted by the mctoll developers [13], lifting gcc compiled executable is not fully tested yet. McSema reuses and extends the optimization passes of the LLVM framework (see [14]) which are generally more correct and effective in processing clang-compiled executables. In fact, we find that IR lifted from gcc-compiled executables is generally more lengthy than those lifted from

clang compiled executables. This observation accounts for the low effectiveness of McSema optimizations in lifting gcc-compiled executables, and may also explain the lower discriminability analysis accuracy for the gcc settings. BinRec shows generally low support for discriminability analysis, which is aligned with findings in Sec. VI-C.

Table XI implies that applying full optimizations (-O3) in the decompilation analysis generates less structured programs, given the higher #goto and greater LOC. Compiler optimizations create extra challenges for reverse engineering. However, in Sec. IX and Appx. C, we will discuss observations that the popular (commercial) decompilers, IDA-Pro and Ghidra [86], exhibit *similar trends* w.r.t. optimized executables. For example, Table XIV in Appx. C shows a comparison of the -O3 and -O0 settings, which reveals that #goto is increased for 5.4 times in IDA-Pro's decompiled C code. In contrast, a comparison of -O3 and -O0 settings in the decompilation evaluation of RetDec-lifted IR code shows an even smaller increase of #goto (3.8 times; smaller is better). Apparently, neither lifters nor (commercial) decompilers can give foolproof solutions to solve C decompilation. Evaluations in this section, however, imply the encouraging and practical value of lifter-driven (particularly RetDec-based) solutions. C code decompiled from McSema-lifted IR and BinRec-lifted IR contain an unusually high #goto for the gcc -O0 setting; this is because several large SPEC programs, including 403.gcc and 400.perlbench, can only be correctly lifted when -O0 is used, which results in many goto statements and thereby increases the average #goto.

IX. Comparison with Binary-Only Tools

We aim to study whether the lifted IR code exhibits good support for downstream security tasks that is *comparable* to that of clang generated IR code. To do so, Sec. VI has studied three tasks that serve as the core building blocks of many downstream security applications. Nonetheless, to directly compare with binary security analysis (without lifting), we compare binary-only tools with lifter-enabled solutions in three tasks, i.e., sanitization, binary diffing, and decompilation. This section summarizes key findings, and we present details in Appx. A, Appx. B, and Appx. C, respectively.

Appx. A compares RetroWrite [43], a binary rewriting framework, and lifters in applying address sanitizer (ASan) [98] to executable. In principle, RetroWrite does not strive to recover variables whereas binary lifters, particularly, RetDec, can (imprecisely) recover variables and function local stacks. This conceptually differentiates RetroWrite from lifters given that RetroWrite only enables coarse-grained stack frame-level ASan insertion. At the empirical level, we compare McSema and RetDec with RetroWrite by inserting ASan checks into the Juliet test programs. Due to the low functionality correctness of RetDec-lifted IR code (see Sec. VI-E), many RetDec-lifted IR programs are mal-functional; it becomes meaningless to benchmark its stack sanitization quality. Out of 3,497 Juliet test cases containing heap vulnerabilities, McSema can correctly lifts 2,187 cases. For these cases, McSema achieves promising heap sanitization accuracy which is comparable with RetroWrite. Nevertheless, the original program stack is not protectable, as McSema uses an emulation stack in its generated IR code. With further enhancement on functionality correctness, McSema has great potential to sanitize executable and detect heap exploitations. In sum, our study in Appx. A, from conceptual and empirical perspectives, illustrates that neither binary-only nor lifter-driven solutions can enable full-fledged sanitization. We also discuss strengths (e.g., cross-platform support) and weaknesses (e.g., performance penalty) of lifter-driven sanitization compared with RetroWrite in Appx. A.

Appx. B compares lifter-enabled binary diffing with the state-of-the-art research tool, DeepBinDiff [46], and the industrial standard tool, BinDiff [2]. The lifter-enabled approaches exhibit encouraging performance in binary diffing, which is comparable to that of DeepBinDiff. This indicates the good potential of lifters for use in security tasks such as malware clustering and CVE/patch searching [120], [124], [121]. In addition, Appx. C assesses the quality of decompiled code, and compares lifter-enabled decompilation with that of (commercial) decompilers, IDA-Pro and Ghidra. These (commercial) decompilers generate more structured code than the lifter-enabled solutions in most settings (see Table XIV), but the latter exhibit reasonable decompilation quality and appealing extensibility, and are cost-free.

X. Discussion

Analyzing Binary Code with LLVM: Are We There Yet? As clarified in Sec. IV, "recompilation" is challenging. However, the research community and industry have provided lifters that can generally pass *functionality* testing and formal verifications [38], [66]. Our study in Sec. VI-E further shows that emulation-style lifters can generate functionality-preserving IR code. These findings have laid a solid foundation to use lifters for code patching, migration, and reuse.

IR lifting, similar with most reverse engineering tasks, is not decidable. Therefore, people may be pessimistic that lifted IR code is invariably lower in quality. However, in daily security tasks, how much of a problem is, for instance, applying some analysis utilities toward the lifted IR code? This SoK paper aims to understand the status quo of lifter-driven static security tasks. We summarize current knowledge and explicate further efforts required to link low-level security analysis with the LLVM framework. We also show that lifters exhibit rather inconsistent support for analysis tasks. Our study enhances the confidence of using lifters in many security tasks related to discriminability analysis, decompilation, and sanitization. While to date one might not expect an "out-of-the-box" usage of lifters for rigorous static analysis, Appx. E explores enhancement using Debin.

Fostering Other Security Tasks. In addition to tasks evaluated in this paper, we also envision using lifters to foster other security tasks. For instance, recent works [84] have studied using lifters in binary-only fuzzing. Given code has been lifted into LLVM IR, it is also a natural extension

to explore bug detection, security patching, and migration using many industrial-strength LLVM-driven solutions [88], [96], [27], [50], [94]. Nonetheless, some of these applications require the correctness of lifted IR code. While our study in Sec. VI-E shows encouraging correctness of emulation-style lifters, this lifting scheme can likely introduce more overhead given that computations are emulated in LLVM IR.

Future Works. As a future research direction, we envision proposing "lifter-oriented" static analysis algorithms. For instance, emulation-style lifting (e.g., `McSema`) uses a global array to emulate the access of the physical memory stack in machine code. This puts scalable field-sensitive static analysis [90] as a basic requirement which supports tracking contents within each array element separately. We plan to benchmark and potentially calibrate standard field-sensitive pointer analysis in analyzing `McSema` lifted IR.

Emulation-style IR (EIR) can promisingly guarantee the functional correctness, but lacks expressiveness. High-level IR (HIR), on the other end of the spectrum, strives to recover high-level code features, but can generate mal-functional code. It thus becomes critical to enhance the quality of EIR and HIR, for which the compiled-IR (CIR) might help. In particular,

Transforming EIR To HIR (EIR $\rightarrow$ HIR). EIR is conservatively lifted in a (mostly) functionality-correct manner. Intuitively, one may explore using a sequence of *semantics-preserving* transformations to gradually change a piece of EIR into HIR, e.g., by finding combinations of certain LLVM passes w.r.t. objective functions like minimizing distances between EIR and HIR. However, we point out that this task, EIR $\rightarrow$ HIR, is *not* fundamentally easier than lifting assembly into EIR. As clarified in **Refinement** of Sec. II, well-known challenges, e.g., variable and type recovery, are required in EIR $\rightarrow$ HIR. In fact, the SecondWrite papers [19], [47] aim to solve EIR $\rightarrow$ HIR with (sound) techniques; see further information about SecondWrite in Sec. XI.

Learning from CIR to Fix HIR (HIR $\leftarrow$ CIR). HIR is lifted in a more expressive manner and is (visually) closer to CIR. To enhance the functional correctness, one may wonder pinpointing and fixing mal-functional code snippets in HIR, by *learning* from corresponding CIR. Recent advances in binary-to-source matching with neural models might be inspiring [119]. In fact, the authors tentatively investigated the feasibility of "HIR $\leftarrow$ CIR"; we find that this direction, although look promising, is much harder than expected. We manually compared some mal-functional HIR with their corresponding CIR. Line-by-line comparison can easily expose inconsistencies. Such inconsistencies, however, are primarily due to uncertainty of reverse engineering, which are not erroneous. Defects in HIR, e.g., ill-lifted local variables, are stealthy. It is difficult, even for human experts, to recognize certain wrong code snippets, let alone machine learning models.

XI. Related Work

Typical applications of binary lifters include optimization [19], [47], [115], code reuse [67], [41], and security analysis [55], [67], [37], [63]. Existing static and dynamic binary lifters either lift an executable into a standard compiler IR (LLVM or GCC) or customized IR. SecondWrite [19], [47] lifts machine code into LLVM IR code and refines it with type inference and symbolic execution. Their proposed technique is demonstrated to generate LLVM IR code of high quality and support code optimization.[1] Egalito [115] proposes the new design of binary layout agnostic IR, which is shown to support fool-proof recompilation on even complex SPEC programs with negligible cost. Inception [37] proposes a "lift-and-merge" process which lifts ARM32 binary code into LLVM IR, and then merges the lifted IR with LLVM bitcode compiled from source code to smoothly enable symbolic execution.

BinRec enhances dynamic lifting using symbolic execution and path merging [18]. Instrew proposes to maintain a code cache [48], which maps machine instruction addresses to the already lifted code fragments during dynamic lifting to avoid repeatedly lifting the same code. LISC [58] automates the generation of assembly to IR lifting rules by learning from how compilers translate IR into assembly instructions. Similarly, Wang et al. [113] propose to learn translation rules for existing dynamic binary translators and use symbolic execution to validate the correctness of the learned rules.

[93] studies the difference of using compiled IR, lifted IR, or assembly code for symbolic execution. Our work has *different focuses* with [93]: [93] compares the symbolic engine of S^2E [35], which is specifically designed for lifted LLVM IR with `KLEE` [29], a symbolic execution engine designed from compiled LLVM IR. In contrast, our systematic study takes one step further by exploring whether analysis facilities in the LLVM compiler framework can be *directly reused* to analyze LLVM IR code lifted by the state-of-the-art binary lifters. Promising support will, to certain extent, push the community to the edge of a breakthrough that allowing to leverage full sets of compiler passes built up over decades to analyze low-level machine code without reinventing the wheel.

XII. Conclusion

We present a study of binary lifters regarding their support for downstream applications. We set up three tasks used by many security applications and study different compilation settings. We also compare lifter-enabled approaches and binary-only solutions over three security tasks. We find that the lifted LLVM IR code exhibits promising support for discriminability analysis and C decompilation, but has much worse support for static analysis. Our findings can provide insights for users and developers that aim to use and enhance lifters.

Acknowledgement

We thank anonymous reviewers and our shepherd, Michael Franz, for their valuable feedback. We also thank Fabian Parzefall and Joseph Nash who provided us with much help and advice in setting up `BinRec`. The research was supported in part by a RGC ECS grant under the contract 26206520.

[1] SecondWrite was seen as commercialized [9]. However, it was confirmed by their engineers that the so-called "SecondWrite" is a malware sandbox. They do not provide any static binary lifter for research study or comparison.

REFERENCES

[1] Research Artifact. https://github.com/monkbai/ir_lifting_data.

[2] BinDiff. https://www.zynamics.com/bindiff.html, 2014.

[3] radare2. http://www.radare.org/r/, 2016.

[4] Bin2LLVM. https://github.com/cojocar/bin2llvm, 2017.

[5] Possible to access higher-level IR in decompilation process. https://github.com/NationalSecurityAgency/ghidra/issues/978, 2019.

[6] RetDec data types propagation. https://github.com/avast/retdec/blob/567c30e3dd4c572fa825d8781ed69652306a961e/src/bin2llvmir/optimizations/types_propagator/types_propagator.cpp, 2020.

[7] RetDec function prototype inference. https://github.com/avast/retdec/blob/567c30e3dd4c572fa825d8781ed69652306a961e/src/bin2llvmir/optimizations/param_return/param_return.cpp, 2020.

[8] RetDec simple types recovery. https://github.com/avast/retdec/blob/567c30e3dd4c572fa825d8781ed69652306a961e/src/bin2llvmir/optimizations/simple_types/simple_types.cpp, 2020.

[9] SecondWrite. https://www.secondwrite.com, 2020.

[10] Translating binaries to LLVM with Revgen. http://s2e.systems/docs/Tutorials/Revgen/Revgen.html, 2020.

[11] Binary Lifter Errors and Our Fixes. https://www.dropbox.com/s/lqskhqrbiunz44j/lifter-sm.pdf?dl=0, 2021.

[12] CVE details. https://www.cvedetails.com/, 2021.

[13] Known issues of mctoll. https://github.com/microsoft/llvm-mctoll#known-issues, 2021.

[14] mcsema optimization passes. https://github.com/lifting-bits/mcsema/blob/0a541eafaf211c1c67c3a4ecbc737792996c6d22/mcsema/BC/Optimize.cpp#L1582, 2021.

[15] Retrowrite codebase. https://github.com/HexHive/retrowrite, 2021.

[16] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and Dae-Hun Nyang. Large-scale and language-oblivious code authorship identification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 101–114, 2018.

[17] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

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

[19] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *EuroSys '13*, 2013.

[20] Shushan Arakelyan, Christophe Hauser, Erik Kline, and Aram Galstyan. Towards learning representations of binary executable files for security tasks. *arXiv preprint arXiv:2002.03388*, 2020.

[21] artemdinaburg. Mcsema demo: Llvm's libfuzzer? https://github.com/lifting-bits/mcsema/issues/131, 2017.

[22] Avast. Retdec. https://github.com/avast/retdec, 2020.

[23] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, pages 5–23. Springer, 2004.

[24] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

[25] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[26] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. NIPS 2018, pages 3588–3600, 2018.

[27] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: a static/symbolic tool for finding good bugs in good (browser) code. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 199–216, 2020.

[28] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 353–368, 2013.

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

[30] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 255–270, 2015.

[31] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108, 2018.

[32] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596. IEEE, 2020.

[33] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with revnic. In *Proceedings of the 5th European Conference on Computer Systems*, pages 167–180, 2010.

[34] Vitaly Chipounov and George Candea. Enabling sophisticated analyses of× 86 binaries with revgen. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 211–216. IEEE, 2011.

[35] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.

[36] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 99–116, 2017.

[37] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-wide security testing of real-world embedded systems software. In *USENIX Sec.*, 2018.

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

[39] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. In *PLDI*, 2016.

[40] Yaniv David, Nimrod Partush, and Eran Yahav. FirmUp: Precise static detection of common vulnerabilities in firmware. In *ASPLOS*, 2018.

[41] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. BISTRO: Binary component extraction and embedding for software security applications. ESORICS. 2013.

[42] Edsger W Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

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

[44] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[45] Yue Duan. Deepbindiff. https://github.com/yueduan/DeepBinDiff, 2020.

[46] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. DeepBinDiff: Learning program-wide code representations for binary diffing. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[47] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. In *PLDI*, 2013.

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

[49] Bauman Erick, Lin Zhiqiang, and Hamlen Kevin W. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

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

[51] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, 2016.

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

[53] Debin Gao, Michael K. Reiter, and Dawn Song. BinHunt: Automatically finding semantic differences in binary programs. ICICS, 2008.

[54] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-miner: Uncovering memory corruption in linux. In *NDSS*, 2018.

[55] Ivan Gotovchits, Rijnard van Tonder, and David Brumley. Saluki: finding taint-style vulnerabilities with static property checking. In *NDSS*, 2018.

[56] Dan Guido. Mcsema author's notes on design goal. https://github.com/microsoft/llvm-mctoll/issues/1#issuecomment-429446832, 2018.

[57] David Gunning. Explainable artificial intelligence (xai). *Defense Advanced Research Projects Agency (DARPA), nd Web*, 2:2, 2017.

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

[59] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1667–1680. ACM, 2018.

[60] SA Hex-Rays. IDA Pro: a cross-platform multi-processor disassembler and debugger, 2014.

[61] Galois Inc. reopt. https://github.com/GaloisInc/reopt, 2020.

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

[63] Anastasis Keliris and Michail Maniatakos Yakdan. ICSREF: A framework for automated reverse engineering of industrial control systems binaries. In *NDSS*, 2019.

[64] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 195–211, 2019.

[65] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *NDSS*, 2018.

[66] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. Testing intermediate representations for binary analysis. In *ASE*, 2017.

[67] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. RevARM: A platform-agnostic arm binary rewriter for security applications. In *ACSAC*, 2017.

[68] J. Křoustek and P. Matula. Retdec: An open-source machine-code decompiler. [talk], July 2018. Presented at Pass the SALT 2018, Lille, FR.

[69] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *PLDI*, 2014.

[70] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *OOPSLA*, 2015.

[71] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*. Citeseer, 2015.

[72] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *NDSS*, 2011.

[73] leepeter2019. Run llvm-mctoll occurs error. https://github.com/microsoft/llvm-mctoll/issues/29, 2019.

[74] Xuezixiang Li, Qu Yu, and Heng Yin. PalmTree: Learning an assembly language model for instruction embedding. *arXiv preprint arXiv:2103.03809*, 2021.

[75] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. *arXiv preprint arXiv:1904.12787*, 2019.

[76] Zhibo Liu and Shuai Wang. How far we have come: Testing decompilation correctness of c decompilers. In *ISSTA*, 2020.

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

[78] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *FSE*, 2014.

[79] McSema. Comparison of machine code to llvm bitcode lifters. https://github.com/lifting-bits/mcsema#comparison-with-other-machine-code-to-llvm-bitcode-lifters, 2020.

[80] Microsoft. LLVM-mctoll. https://github.com/Microsoft/llvm-mctoll, 2020.

[81] Microsoft. Mctoll. https://github.com/microsoft/llvm-mctoll, 2021.

[82] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[83] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[84] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[85] Coseinc Nguyen Anh Quynh. Capstone. http://www.capstone-engine.org/, 2021.

[86] National Security Agency (NSA). Ghidra. https://www.nsa.gov/resources/everyone/ghidra/, 2018.

[87] Trail of Bits. McSema. https://github.com/lifting-bits/mcsema, 2018.

[88] Trail of Bits. Polytracker: An LLVM-based instrumentation tool for universal taint tracking, dataflow analysis, and tracing, 2021.

[89] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NIPS*, 2019.

[90] David J Pearce, Paul HJ Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(1):4–es, 2007.

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

[92] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. TREX: Learning execution semantics from micro-traces for binary similarity. In *IEEE Symposium on Security and Privacy (SP)*, 2020.

[93] Sebastian Poeplau and Aurélien Francillon. Systematic comparison of symbolic execution systems: intermediate representation and its generation. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 163–176, 2019.

[94] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 49–64, 2015.

[95] rev.ng Srls. Rev.ng. https://rev.ng/, 2018.

[96] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. PhASAR: An inter-procedural static analysis framework for c/c++. In *TACAS (2)*, pages 393–410, 2019.

[97] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. Using logic programming to recover c++ classes and methods from compiled executables. In *CCS*, 2018.

[98] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, 2012.

[99] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[100] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

[101] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019.

[102] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266. ACM, 2016.
[103] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.
[104] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *OOPSLA*, 2016.
[105] sunlv. Fail to build klee maze example. https://github.com/lifting-bits/mcsema/issues/569, 2019.
[106] testhound. Mctoll crashes when running raising arm binary. https://github.com/microsoft/llvm-mctoll/issues/67, 2020.
[107] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. Software protection on the go: A large-scale empirical study on mobile app obfuscation. In *ICSE*, 2018.
[108] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.
[109] Shen Wang, Zhengzhang Chen, Xiao Yu, Ding Li, Jingchao Ni, Lu-An Tang, Jiaping Gui, Zhichun Li, Haifeng Chen, and Philip S Yu. Heterogeneous graph matching networks for unknown malware detection. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 3762–3770. AAAI Press, 2019.
[110] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *USENIX Sec.*, 2015.
[111] Shuai Wang, Pei Wang, and Dinghao Wu. Uroboros: Instrumenting stripped binaries with static reassembling. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 236–247. IEEE, 2016.
[112] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural networkand flow-augmented abstract syntax tree. *arXiv preprint arXiv:2002.08653*, 2020.
[113] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Enhancing cross-isa dbt through automatically learned translation rules. *ACM SIGPLAN Notices*, 53(2):84–97, 2018.
[114] Brian Wickman, Hong Hu, Insu Yun Daehee Jang, JungWon Lim Sanidhya Kashyap, and Taesoo Kim. Preventing use-after-free attacks with fast forward allocation. 2021.
[115] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 133–147, 2020.
[116] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.
[117] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*, 2015.
[118] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1145–1152, 2020.
[119] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. CodeCMR: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems*, 33, 2020.
[120] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *USENIX Security*, 2018.
[121] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *Usenix Security*, 2021.
[122] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. STOCHFUZZ: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. 2021.
[123] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. pages 175–186, 2013.
[124] Lei Zhao, Yuncong Zhu, Jiang Ming, Yichen Zhang, Haotian Zhang, and Heng Yin. PatchScope: Memory object centric patch diffing. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 149–165, 2020.
[125] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

APPENDIX

A. Binary Code Sanitization

Address sanitizer (ASan) [98] inserts sanitizer checks to detect software defects such as buffer overflow [101]. It instruments each memory access to validate memory addresses with sanitizer checks. ASan also uses a runtime library to hook memory allocation/free and to create poisoned "redzones" for each allocated memory region to detect memory errors.

Taxonomy of Memory Protection Capability. The LLVM framework has provided ASan as a standard utility. We enable ASan by compiling lifted IR using `clang` and with the `-fsanitize=address` option. In addition, we reuse RetroWrite [43], a binary-only rewriting framework to insert sanitizer checks into binary code. RetroWrite leverages Capstone [85] for disassembling. It facilitates relocation symbol recovery and ASan check insertion. We compare RetroWrite with `McSema` and `RetDec` given their distinct and representative code lifting schemes, i.e., emulation-style lifting vs. high-level lifting (Sec. III). We follow the notation used in the RetroWrite paper, and report to what extent different memory regions are protected by ASan in Table XIII. As a binary-only solution, RetroWrite can redzone heap but does not strive to recover global or local variables. RetroWrite thus instruments stack objects at the stack frame granularity, which may miss bugs when the overflow is contained within the frame [43]. Recall `McSema` uses emulation-style lifting, which means that the recovered function local stack no longer corresponds to the original stack. Hence, we deem its lifted IR does not facilitate redzoning stack, but protects heap and globals. Our previous study has shown that `RetDec` (partially) recovers variables and program stacks which are seen to be similar with the `clang`-generated LLVM IR code. In principle, `RetDec`-lifted IR code can enable the securing of stacks and global memory regions, thereby overcoming the limits of binary-only tools. Nevertheless, its variable recovery is *inaccurate*, which means that its memory protections are not flawless; we use "partial" in Table XIII to denote this aspect.

Experiments. We use the Juliet test suite that was benchmarked in the RetroWrite paper, which is a collection of test programs containing stack and heap memory vulnerabilities. Each test program has a "good" variant without a vulnerability and a "bad" variant with a vulnerability (generated by slightly modifying the "good" variant). Hence, tools can be evaluated regarding errors reported on the bad variants while not flagging errors on the good variants.

We report the bug detection rates of ASan-enabled programs in Table XII, and compare these with the rates reported for RetroWrite (reusing results in its paper). For each Juliet test case, we first compare the outputs of the ASan-enabled good variant with its original good variant using its shipped test

TABLE XII
VULNERABILITY DETECTION RATES. "FALSE POSITIVES" ARE OMITTED AS THEY ARE ALL ZERO FOR ALL SETTINGS.

	clang & ASan	RetroWrite	McSema & ASan	McSema & ASan (heap vulnerability)	RetroWrite & ASan (heap vulnerability)	RetDec & ASan
#Functionality Correct Cases	5,914	5,912	6,350	2,187	3,497	692
True Positive	4,489	3,257	2,097	1,862	3,124	0
True Negative	5,914	5,912	6,350	2,187	3,497	692
False Negative	1,382	2,614	4,253	325	355	692
Recall	76.5%	55.5%	33.0%	85.1%	89.3%	0%

TABLE XIII
OVERVIEW OF REDZONE POLICY AS ENABLED BY STANDARD ASAN, RETROWRITE, AND LIFTED LLVM IR. "PARTIAL" CONSERVATIVELY DENOTES THAT VARIABLE RECOVERY IS NOT ACCURATE, AND FUNCTIONALITY MAY BE BROKEN IN THE FIRST PLACE.

Memory Object	ASan	RetroWrite	McSema	RetDec
Heap	Full	Full	Partial	Partial
Stack	Full	Lower granularity	NA	Partial
Global	Full	NA	Partial	Partial

inputs; if the outputs of two good variants are equal (see the second row of Table XII), we then test if ASan checks in the corresponding bad variant can capture the memory vulnerability. A false positive means that ASan reports to capture the vulnerability in a good variant whereas false negative means that ASan misses to capture the vulnerability in a bad variant. Given that McSema uses emulation-style stacks, we exclude Juliet test programs containing stack vulnerabilities and report results derived from the remaining ASan in the 5th column of Table XII (heap vulnerabilities). Similarly, we also run the released RetroWrite tool [15] on these heap vulnerability cases and report the results in the sixth column.

clang-inserted ASan (the second column in Table XII) has the highest recall rate, and the remaining false negative cases (1,382) arise because some vulnerabilities in the Juliet dataset are not designed to benchmark ASan. RetroWrite (the third column) shows a good recall rate, which is comparable to that of clang-inserted ASan. More importantly, RetroWrite and McSema on heap vulnerability cases yield promising and comparable recall rates (85.1% vs. 89.3%); however, McSema generates 1,310 lifted IR programs which are mal-functional (see the second row of Table XII). It thus becomes meaningless to benchmark ASan over those 1,310 programs. In addition, our manual study shows that some remaining false negatives are due to bugs are not triggerable in 64-bit PIC code. Note that RetroWrite can only process 64-bit PIC code whereas McSema is a versatile *cross-platform* binary lifter that can process binary code on 32-bit/64-bit x86, ARM, and SPARC architectures. In sum, given that many heap-related vulnerabilities, such as use-after-free bugs, remain unsolved [12], [71], [114], McSema has appealing utility, as on the correctly-lifted IR programs, it shows heap-sanitization performance that is comparable to that of RetroWrite, the state-of-the-art binary-only solution.

In addition, McSema's emulation-style lifting can generate slower code than binary-only solutions. In all, McSema carries large runtime performance overhead as computations are emulated in its lifted IR code. Nonetheless, for Juliet test cases, we report the extra runtime cost is negligible. Our experiments show that McSema +ASan introduces about 40% extra slowdown whereas RetroWrite +ASan is about 37%. This is reasonable: while Juliet test cases contain comprehensive sets of vulnerabilities, these test programs are not so complex. Nevertheless, we note that complex executable may impose major challenge for both McSema and RetroWrite: the emulation-style lifting of McSema can likely introduce notable performance penalty, whereas the "re-assemable disassembling" heuristics [110] employed by RetroWrite might be broken (i.e., generating mal-functional code).

Our manual study confirms that in RetDec-lifted IR, ASan checks are inserted in a generally comprehensive way, similar to those in clang-generated IR. However, almost all LLVM IR lifted by RetDec fails to retain functional correctness. After excluding Juliet test programs that have erroneous outputs, we have 692 remaining programs. We report that vulnerabilities in all the "bad" variants of these 692 programs are missed, resulting in 692 false negatives in Table XII. This evaluation reveals the lack of binary lifters capable of delivering full-fledged sanitization support. Heap memory regions can be protected based on McSema-lifted IR, which exhibits accuracy comparable to that of RetroWrite. In contrast, the sanitization of stacks is dependent on the functionality correctness of RetDec-lifted IR code, which requires major improvements.

TABLE XV
DEEPBINDIFF BINARY DIFFING COMPARISON RESULTS ON POJ-104 RESULTS OVER DIFFERENT SETTINGS.

	BinDiff	DeepBinDiff	RetDec
gcc -O0	70.2%	66.3%	77.7%
gcc -O3	54.8%	68.3%	80.1%
clang -O0	57.7%	73.1%	81.0%
clang -O3	59.6%	66.3%	78.2%

B. Binary Diffing Comparison

We also compare our binary similarity analysis results with those generated by the state-of-the-art binary diffing tool, DeepBinDiff [46], and by the industrial standard binary diffing tool, BinDiff [2]. The released DeepBinDiff implementation [45] is highly convenient to use, as it does not require a pre-trained model. Given two executable files, it launches an on-the-fly training process and conducts basic block level matching.

Recall our discriminability model used in Sec. VI-C is on the *whole program-level*, whereas the DeepBinDiff

TABLE XIV
AVERAGE STRUCTUREDNESS AND LOC PER USER-DEFINED FUNCTION OF SPEC PROGRAMS.

Tools		clang	McSema	RetDec	BinRec	IDA-Pro	Ghidra
gcc -O0	#goto	NA	15479.3	1661.3	6661.4(12978)	856.9	332.3
	LOC	NA	282.7	142.3	245824.2(478924.3)	68.4	60.0
gcc -O3	#goto	NA	8023.2	3766.4	1820(20078.2)	5304.7	2554.9
	LOC	NA	595.1	374.3	61677(680419.3)	122.1	90.3
clang -O0	#goto	3188.4	6026.5	1709.6	6344.1(27894.4)	865.8	653.8
	LOC	44.3	247.6	168.7	70947.1(311947.6)	65.2	61.4
clang -O3	#goto	6589.9	9367.0	8971.5	1635(3448.7)	4016.9	2211.2
	LOC	112.9	554.6	276.6	36256.7(76476.1)	109.8	85.9

implementation [45] delivers *basic block-level* matching. BinDiff also performs a program-wide comparison. To launch a fair comparison, we design the following task: given a pair of POJ-104 programs p_1 and p_2 written to solve the same programming assignment, a program p_3 for an irrelevant assignment is randomly selected. We then use DeepBinDiff to compare p_1 with p_2 and p_3, respectively. Let the number of matched basic blocks be $bb_{1,2}$ and $bb_{1,3}$, we deem a *correct* match where $bb_{1,2} \geq bb_{1,3}$, and vice versa. For our discriminability model (the RetDec column) and BinDiff, we check whether the program-wise similarity between p_1 and p_2 is higher than that of p_1 and p_3. The accuracy scores are reported in Table XV. We launch this evaluation on the test split of POJ-104, whereas the discriminability model is trained using RetDec-lifted IR code over the POJ-104 training split. We compute and report the accuracy scores in Table XV.

RetDec and DeepBinDiff outperform BinDiff for most settings, and RetDec manifests relatively higher accuracy compared with DeepBinDiff. We interpret the results as reasonable: one needs to train our discriminability model whereas the released implementation of DeepBinDiff performs on-the-fly training over a pair of binary code. The accuracy of DeepBinDiff can be further improved given that a pre-trained model is used in the DeepBinDiff paper. We also note that most binary diffing works [46], [44], [92] explore a potentially easier task: the comparison of a pair of executables compiled from the *same* program using different compilation settings or different versions. In contrast, the experiment we perform follows the setting in [26], [82] to compare executables of two programs implementing the same programming assignment.

We conclude that DeepBinDiff is generally suitable for daily security analysis and binary diffing tasks, given its promising performance [46] and convenience. For heavyweight program-wise binary diffing, another presumably promising option is to employ binary lifters and LLVM-level representation learning tools like ncc.

C. Decompilation

We measure the decompilation quality of (commercial) decompilers and compare the results with our findings in Sec. VI-D. We use IDA-Pro and Ghidra, two popular commercial and free decompilers, to decompile executables into source code. Table XIV reports the evaluation results. Note that since our cross-compiler and cross-optimization evaluation in Sec. VIII has discussed and cross-compared different lifters in supporting decompilation, this section only compares RetDec with two (commercial) binary decompilers.

As expected, both IDA-Pro and Ghidra perform very well in terms of C/C++ decompilation. Although in the gcc -O3 setting, RetDec has less #goto than IDA-Pro, the C code decompiled by IDA-Pro/Ghidra manifests generally better structuredness for most settings. This is reasonable; IDA-Pro is a mature commercial product, and Ghidra is developed by NSA with vast resources. The engineering quality of the RetDec decompiler has been reported to be slightly lower (containing more bugs) than that of IDA-Pro and Ghidra [77]. Table XIV and our study in Sec. VI-D show that lifted IR and compiled IR exhibit comparable support for decompilation. Studies in this section show that LLVM IR-to-C decompilation requires further improvement. Nevertheless, the LLVM IR-to-C approach is a *free* and *highly extensible* solution based on the LLVM ecosystem. In contrast, it appears that the customized IR of Ghidra would be difficult (or impossible) to modify [5].

TABLE XVI
BINARY LIFTING RESULTS FOR THE ARM64 SETTING. WE PRESENT TOTAL TEST CASES AND THE NUMBER OF FAILED LIFTING CASES.

Dataset	SVF	POJ-104	SPEC
Total test cases	84	49,058	9
RetDec	0	0	0
McSema	0	28	0

TABLE XVII
BINARY LIFTING RESULTS FOR THE CROSS-COMPILER AND OPTIMIZATION SETTING. WE PRESENT TOTAL TEST CASES AND FAILED LIFTING CASES.

Dataset		POJ-104	SPEC
gcc -O3	Total test cases	49,275	9
	RetDec	205	2
	McSema	35	3
	mctoll	28,185	9
	BinRec	18.651	4
gcc -O0	Total test cases	49,275	9
	RetDec	60	1
	McSema	181	0
	mctoll	33,283	9
	BinRec	20,741	3
clang -O3	Total test cases	49,154	9
	RetDec	23	1
	McSema	33	4
	mctoll	34,283	9
	BinRec	15,079	5

D. Binary Lifting

Table XVI reports binary lifting results in accordance with the cross-platform evaluation in Sec. VII. McSema has a few (less than 1%) lifting failures in the POJ-104 test cases. Similarly, Table XVII reports the binary lifting results in accordance with the cross-compiler/optimization evaluation in Sec. VIII. Again, mctoll cannot process executable compiled from C++ code, and therefore, it fails to lift a large number of POJ-104 cases. It also fails to lift all SPEC test programs. We have reported our findings to the mctoll developers. RetDec and McSema perform generally well by making fewer lifting errors.

Note that when launching cross-compiler and cross-optimization evaluation, we use gcc (ver. 7.5.0) whereas in the BinRec paper, gcc (ver. 4.8.4) is evaluated. We find a number of opcodes that are not supported by the current implementation of BinRec, which explains the higher lifting failures of BinRec for the gcc cases. In comparison, BinRec has only 9,668 failures for clang (-O0) compiled binary code (Table V). Also, when lifting POJ-104 programs, we set a shorter timeout threshold (2 minutes), which induces more cases that cannot be finished within this threshold. Nevertheless, our findings in Sec. VIII should not be affected given that we still successfully lifted over 20K executables for each compilation setting.

TABLE XVIII
ENHANCING POINTER ANALYSIS RESULTS OF RETDEC.

	MustAlias	MayAlias	NoAlias
RetDec	0.0%	0.0%	88.9%
RetDec + Debin	0.0%	0.0%	88.9%
RetDec + Debin$^+$	57.1%	100.0%	94.4%
RetDec + debug info	71.4%	100%	94.4%
RetDec + manual fix	100.0%	100.0%	100.0%

E. Pointer Analysis Enhancement

Overall, pointer analysis has the lowest accuracy of the three downstream applications evaluated in Sec. VI and Sec. VII, which reveals a considerable gap between the performance of compiled IR code and lifted IR code. In contrast, binary lifters, particularly RetDec and mctoll, exhibit satisfactory discriminability analysis and C decompilation accuracy, which is close to compiled IR code. This section explores practical strategies to enhance the accuracy of pointer analysis. We use the 16 fixable cases identified in the "manual fix" conducted in Sec. VI-B (see Table VII) and focus on enhancing RetDec, as it outperforms other lifters.

As reported in Table VII, variable types primarily contribute to the correctness of pointer analysis. Hence at this step, we first use binary code with debug information available (using -g option when compiling with clang) and explore whether pointer analysis results can be improved. The fifth row of Table XVIII reports the evaluation results. RetDec leverages the debug information to refine the recovered variables, variable types, and function prototypes. This can effectively promote the pointer analysis results. On the other hand, our manual comparison between the fifth row and the last row (manual fix) shows that RetDec still makes considerable errors in recognizing global variables (even with the presence of debug information and symbol tables). Considering the following C statement and its corresponding lifted IR statement:

```
int *p = &x; // p and x are both global var.
@p = i32* inttoptr (i32 134520892 to i32*)
```

where 134520892 is the memory address of x in assembly. Address 134520892 was not symbolized into a LLVM local variable representing x. We confirm that all global pointer initialization (following the above pattern) are ill-handled by RetDec, incurring errors in the fifth row of Table XVIII. We have reported this bug to the RetDec developers.

Debug and symbol information in stripped executable can be recovered by Debin [59]. Debin recognizes variables with a randomized tree classifier and employs a probabilistic graphical model to make joint predictions. We reuse the author released Debin pre-trained on x86 executables to predict debug and symbol information on stripped executable. We then re-run SVF on the enhanced binary code. Unfortunately, as shown in the third row of Table XVIII, Debin can barely enhance pointer analysis (we have excluded false positives). We find that for the SVF test cases, Debin is unable to correctly recover function prototypes. We further augment the output of Debin, by manually fixing all the function argument types and re-running SVF. As a result, the local variables and their types are recovered by Debin, while function prototypes are recovered with our manual efforts. The fourth row (Debin$^+$) reports the evaluation results. Debin shows promising accuracy in recovering local variables and types; our manual inspection shows that the LLVM IR code enhanced by Debin is close to directly lifting binary code with debug information available. Overall, we interpret that study in this section sheds light on practical solutions to enhance the quality of lifted IR code, by first using (and augmenting) debug information recovery tools.