DIATOM: Polylithic Binary Lifting with Data-flow Summaries and Type-aware IR Linking

ANSHUNKANG ZHOU, Hong Kong University of Science and Technology, China
CHARLES ZHANG, Hong Kong University of Science and Technology, China

Binary lifting, which translates binary code into LLVM intermediate representations (IRs) through iterative IR transformations for recovering high-level constructs from low-level machine features, is the cornerstone of many binary analysis systems. Therefore, the scalability and precision of the upper layer analysis could be greatly affected by the underlying binary lifting. However, all existing binary lifters still suffer from severe performance problems in that they require much time to handle extremely large binaries, which becomes a barrier to achieving the expected performance gains in various analyses and hinders them from meeting the requirement of quick response in modern continuous integration pipelines. We found that the root cause of the scalability issue is the inherent "monolithic" design that performs all lifting stages on a single LLVM module, which entails a global environment that enforces sequential dependences between any two transformations on IRs, thus limiting the parallelism.

This paper presents DIATOM, a novel parallel binary lifter powered by a new "polylithic" design, which decomposes the monolithic LLVM module into partitions to perform fully parallelized binary lifting. In the meantime, it leverages light-weight data-flow summaries and type-aware IR linking to avoid soundness loss caused by separating dependent code fragments. Large-scale experiments on 16 real-world benchmarks whose sizes range from dozens of megabytes (MBs) to several gigabytes (GBs) show that DIATOM achieves an average speedup of 7.45× and a maximum speedup of 16.8× over a traditional monolithic binary lifter, while still maintaining the lifting soundness. DIATOM can complete the translation for the `Linux Kernel` binary within only 10 minutes, which significantly accelerates the overall binary code analysis process.

CCS Concepts: • **Security and privacy** → **Software reverse engineering**.

Additional Key Words and Phrases: Binary lifting

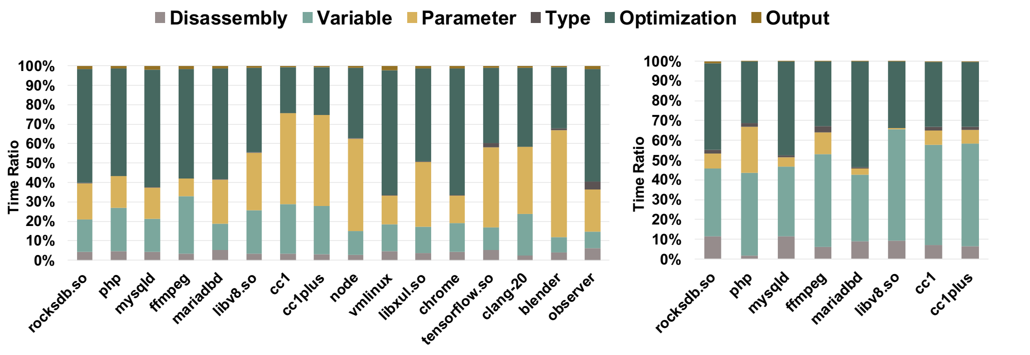
ACM Reference Format:
Anshunkang Zhou and Charles Zhang. 2026. DIATOM: Polylithic Binary Lifting with Data-flow Summaries and Type-aware IR Linking. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 98 (April 2026), 32 pages. https://doi.org/10.1145/3798206

1 Introduction

Binary lifting [6, 13, 31, 34, 37, 38, 92, 108, 130], which translates the human-unreadable machine code into high-level LLVM intermediate representations (IRs) [128], is the prerequisite for reasoning about the semantics or behaviors of binary files [90, 130], and is the cornerstone of many binary-related analysis tasks such as malware analysis [73, 74], binary rewriting [32, 38–40, 84, 92], and vulnerability discovery [29, 130]. Therefore, the scalability and precision of the upper layer analysis could be greatly affected by the underlying binary lifting [66]. While improving the precision of binary lifters to facilitate better binary code analysis has been extensively studied [31, 38, 92, 130], the scalability problem has always been overlooked.

Authors' Contact Information: Anshunkang Zhou, Hong Kong University of Science and Technology, China, azhouad@cse.ust.hk; Charles Zhang, Hong Kong University of Science and Technology, China, charlesz@cse.ust.hk.

ACM 2475-1421/2026/4-ART98
https://doi.org/10.1145/3798206



(a) Plankton (b) RetDec

Fig. 1. Runtime breakdowns on different lifting stages of two state-of-the-art binary lifters Plankton [130] and RetDec [57] for processing 16 large binaries used in § 8 (RetDec only successfully processes 8 binaries).

Binary files are typically generated by the compilation/linkage process and normally lack symbol-/modularization information. Lifting binaries into intermediate languages such as the LLVM IR requires abstracting away from low-level features of the assembly language itself and recovering high-level constructs available in most common programming languages [28] such as inter-procedural control-flow graphs (ICFGs) and variable entities. The process mainly relies on three crucial analyses: control-flow analysis [21, 63, 117], data-flow analysis [20, 49], and type inference [26, 85]. The control-flow analysis aims to recover the inter-procedural control-flow graph (ICFG) by identifying instructions, blocks, and function boundaries from raw bytes inside the original binary file, and removing direct address references in control-flow transfer instructions (e.g., `jmp`). The data-flow analysis aims to reconstruct high-level language features such as variables, parameters, and function return values and to remove hardware references from the code, such as registers, data sections, and the stack. The type inference aims to assign high-level data types (e.g., pointers and structures) to the recovered high-level code entities such as variables and parameters.

Due to the information loss during compilation, different from source code files that are naturally modular and can be analyzed separately and incrementally, the binary code is often monolithic, i.e., all machine instructions and data are tightly integrated as a whole [79]. Lifting monolithic binary code into intermediate representations (IRs) usually requires whole program analysis on the entire binary memory space to maintain the translation *soundness* [10, 14, 22, 75, 109, 110, 117]. For example, since control transfer instructions can refer to any offsets inside the binary, control-flow recovery has to consider every possible machine address for indirect jumps [98]. In practice, achieving perfect soundness is provably *undecidable* [10, 82, 109], meaning that one cannot build a lifter that guarantees a fully faithful translation for every possible binary. Existing lifters therefore preserve a subset of the original binary's behaviors under controlled assumptions [82], which is not a weakness but a pragmatic design principle. In order to retain as much information as possible, when lifting from the original binary to the IR, existing binary lifters such as Plankton [130], RetDec [57, 66], mctoll [92], and McSema [31, 37] all follow a so-called "**monolithic**" design, which performs lifting stages such as disassembly, variable recovery and code optimizations [13, 34, 57, 92, 130] on a single LLVM module sequentially [50, 67, 67, 93].

1.1 Problem

However, the monolithic design inherently limits the scalability of existing lifters since the lifting process that consistently transforms (modifies) IR statements is difficult to parallelize on a single

LLVM module. Every LLVM module $\mathcal{I}$ contains an environment $\mathcal{E}$ with a bunch of information, such as type mappings and global def-use chains on constants [58, 128]. $\mathcal{E}$ is shared globally by all IR statements inside $\mathcal{I}$ and will be updated by each transformation $\mathcal{T}$ on the $\mathcal{I}$ ($\mathcal{E} \hookrightarrow^{\mathcal{T}} \mathcal{E}'$), which leads to sequential dependences between transformation operations (i.e., $\mathcal{T}_i$ depends on $\mathcal{T}_{i-1}$), thus hindering the parallelism in monolithic design-based binary lifters. Hence, existing lifters are hard to scale and suffer from severe performance problems when handling extremely large real-world binaries. For example, the state-of-the-art binary lifter PLANKTON [130] still takes about 2 hours to lift the 2GB `Chrome`. RETDEC [57] even takes over 4 hours to lift the 472MB `libv8.so`.

The scalability of the underlying binary lifting is crucial for the practicality of upper layer analysis because it acts as the initial and one of the most frequently invoked components in many real-world scenarios, such as static analysis [130] and binary rewriting [13, 38, 79]. For example, lifting time is important for binary rewriters [13, 38, 79] because rewriting could be performed iteratively for more optimized rewritten binaries [72] or higher code coverage [13]. Even worse, although modern static analysis has been able to finish checking one million lines of code within only 15 minutes using advanced parallelism/distributed algorithms [27, 104, 106, 131] or on-demand techniques [94, 95], existing binary lifters could still require a few hours to translate binaries on such a large scale, which becomes a barrier to achieving the expected performance gains.

1.2 Basic Idea

Our key insight to solve the scalability problem is that the majority of lifting time is spent on stages that assume a fixed inter-procedural control-flow graph (ICFG), which can be accelerated through function-level parallelism. All binary lifters [13, 31, 34, 37, 57, 92, 130] rely on a single disassembly [80] step for the ICFG recovery and all the remaining lifting stages are performed on the initial ICFG without modifying it. To better illustrate our insight, we studied the breakdown of time consumption on different lifting stages of existing binary lifters for processing 16 large binaries. Figure 1 shows the results of two state-of-the-art binary lifters PLANKTON [130] and RETDEC [57, 66]. The results for RETDEC in Figure 1b include only 8 binaries because it crashes on the others (no valid LLVM IRs produced). The results demonstrate that the disassembly stage, which could modify the ICFG, only accounts for a very small portion of the total lifting time (as low as 5.95% on average), and the majority of the lifting time (as high as 94.05%) assumes a fixed ICFG.

Therefore, the basic idea is what we refer to as a "**polylithic**" design, which splits all functions inside the initial IR produced by the disassembly into a set of individual IR partitions $\{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_n\}$ with independent environments $\{\mathcal{E}_1, \mathcal{E}_2, ..., \mathcal{E}_n\}$ to perform the remaining lifting stages in parallel, and those partitions are later linked together to produce the final IR output $\mathcal{I}_{link}$.

1.3 Challenge

However, several lifting stages require reasoning about inter-procedural data-flows between functions, such as parameter passing [92] and stack memory sharing [92, 130]. Therefore, a splitting strategy that considers code dependences such as caller-callee relations limits the scalability [96], while directly splitting the monolithic IR and parallelizing the translation of individual functions could be unaware of inter-dependences and cause unsound lifting results.

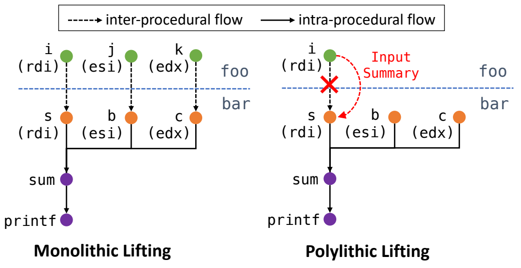
For example, consider the disassembled assembly code in Figure 2b. Monolithic lifting could recover an inter-procedural data-flow from `foo`'s first parameter `i` to `bar`'s first parameter `s` (`i` $\rightsquigarrow$ `s`) by analyzing the two function bodies, and produces the PDG shown in the left-hand side of Figure 2c. On the one hand, if polylithic lifting does not separate `bar` and `foo` due to their caller-callee relations, the parallelism will be limited. On the other hand, if `bar` and `foo` are split into individual partitions, the lifter will be unaware of operations inside the caller or the callee, thus could miss the data-flow between `i` and `s`, as shown by the right-hand side of Figure 2c.

```c
1  void bar(St* s,int b,
2           int c,int d,
3           int e,int f,
4           int g)
5  {
6    int sum = s->a+b+c
7                  +d+e
8                  +f+g;
9    printf("%d",sum);
10 }
11 void foo(St* i,int j,
12          int k)
13 {
14   bar(i,j,k,1,2,3,4);
15 }
```

(a) C source code.

```
1  ; bar
2    add   esi, edx ; b+c
3    add   esi, ecx ; d
4    add   esi, r8d ; e
5    add   esi, r9d ; f
6    add   esi, [rsp+8] ;g
7    add   esi, [rdi] ;s->a
8    mov   edi, 0x76 ; "%d"
9    jmp   printf
10 ; foo
11   mov   [rsp], 4 ; g
12   mov   ecx, 1   ; d
13   mov   r8d, 2   ; e
14   mov   r9d, 3   ; f
15   call  bar
```

(b) Assembly code.



(c) Simplified PDGs of lifted IRs.

Fig. 2. Example source code, the corresponding assembly code disassembled from its compiled binary, and the program dependence graphs (PDGs) generated by monolithic/polylithic lifting.

1.4 Our Technique

This paper presents DIATOM, a novel parallel binary lifter powered by a new polylithic design, which decomposes the monolithic IR into individual partitions to parallelize the lifting process. Our key idea to address the paradox between scalability and soundness is two-fold.

First, inspired by summary-based inter-procedural static analysis, which reuses information within a procedure as summaries for better efficiency [36, 62, 88, 94, 95, 105, 120] and for enabling parallelism [12, 50, 96], we propose to leverage a special kind of function summaries to realize separate lifting of individual functions in the presence of incomplete function bodies. For example, on the right-hand side of Figure 2c, we leverage pre-generated input summaries to maintain the data-flow between `i` and `s` in the absence of explicit function definitions. It is possible to use summaries that are pre-generated on the disassembled IR with only low-level machine features to facilitate transformed IR with higher-level abstractions because data-flows remain the same for both low-level machine code and high-level lifted IR, but in different forms. Previous summary-based approaches [36, 94, 95, 103, 105, 106] that compute the local side effects of a function on its formal parameters and return values are not applicable to polylithic lifting due to three reasons. First, high-level constructs such as parameters are not directly accessible at the binary level and require further recovery. Second, binary code has implicit side effects that cannot be retrieved by intra-procedural analysis. For example, for the call site at Line 15 in Figure 2b, the first parameter that should be passed by `rdi` does not have explicit operations inside `foo`. Third, they compute heavy-weight information such as points-to sets [94], which are still too expensive and unnecessary to serve as the pre-analysis for binary lifting. Our key insight is that light-weight function summaries are sufficient for the data-flow recovery phase in binary lifting because (1) data-flows on low-level machine code are implemented by registers that are globally defined without aliasing so that they can be analyzed through cheap forward data-flow tracking, (2) and the summary generation process does not modify IR statements and thus can be fully parallelized. Specifically, we perform forward traversal on the function body to extract data-flow summaries for the stack pointer register and registers for parameter passing, which are further refined by fixed-point propagation on the call graph for handling implicit data-flows. The generated summaries are then used by the parallel threads as global information to maintain the data-flow soundness during information recovery.

The above summary-based method still cannot preserve the type soundness because the type inference is performed on code entities (e.g., variables and function parameters) created during the lifting process, which cannot be analyzed in advance. The key insight is that since type inference on binary code works by aggregating how a specific value is used inside the program, such aggregation can also be performed separately. Therefore, instead of preserving type soundness during parallel lifting, the second part of our idea performs post-processing during the IR linking step, which leverages the subtyping relations between different types to reconcile code entities in different partitions, such that the type soundness can be guaranteed.

Novelty. We would like to highlight three key novelties. First, we propose a new polylithic design to improve the scalability of binary lifters, which has not been explored before. The second novelty lies in infusing the summary-based methods and subtyping relations at different phases of polylithic design, together with several modifications to resolve the paradox between scalability and soundness, thus exposing more parallelism. Third, the proposed design also embraces a new asynchronous way to perform binary lifting, which paves the way for further improvements in binary lifters.

Scope. The scope of this work is confined to improving the scalability of binary lifting by accelerating the translation of binary machine code into LLVM intermediate representations (IRs). Our approach focuses on enhancing efficiency through parallelization, without relying on any symbol or debugging information from the binary. It does not aim to improve translation precision or recover higher-level semantics beyond what existing lifters provide. Instead, the goal is to enable scalable binary lifting for extremely large binaries, serving as a high-performance foundation for subsequent binary analysis tasks.

We implemented Diatom and evaluated it with 16 large real-world benchmarks whose sizes range from dozens of megabytes (MBs) to several gigabytes (GBs). The results show that Diatom achieves an average speedup of 7.45× and a maximum speedup of 16.8× while still maintaining the lifting soundness. In summary, this paper makes the following contributions:

- We propose a novel parallel binary lifter that follows a new polylithic design.
- We propose to use light-weight data-flow summaries and type-aware IR linking to resolve the paradox between scalability and soundness in polylithic binary lifting.
- We conduct large-scale experiments to show that Diatom runs much faster than existing binary lifters by achieving 7.45× speedup on average while still being soundness-preserving.

2 Design Overview

Algorithm 1: Polylithic Parallel Binary Lifting

```
Input: A binary program B, and the number of available threads n
Output : Lifted LLVM IR $\mathcal{I}_{link}$
1  Function PolylithicLifting(B):
2      $\mathcal{I}_{dis} \leftarrow$ Disassembly(B)
3      $\psi_{\mathcal{S}} \leftarrow$ ComputeSummary($\mathcal{I}_{dis}$)                ▷ data-flow summary generation
4      $\mathcal{I}_{link} \leftarrow$ empty LLVM IR module                                  ▷ output IR
5      $\psi_d \leftarrow$ DistributeFunctions($\mathcal{I}_{dis}, n$)          ▷ distribute functions into groups
6      foreach index i in $\psi_d$ do in parallel
7          $\mathcal{I}_i \leftarrow$ Clone($\mathcal{I}_{dis}, i, \psi_d$)                       ▷ IR partitioning
8          performing other lifting steps on $\mathcal{I}_i$ using $\psi_{\mathcal{S}}$                ▷ lifting
9          Link($\mathcal{I}_{link}, \mathcal{I}_i$)                                           ▷ link partitions
10     return $\mathcal{I}_{link}$
```

The key design goal of Diatom is to achieve ultimate scalability in binary lifting by following a novel "polylithic" design instead of the conventional "monolithic" one. In the meantime, it does not sacrifice the lifting soundness (*soundness-preserving*) achieved by the traditional monolithic lifting that performs whole program analysis, i.e., the result of polylithic lifting should recover as much information as the result of the monolithic one. Therefore, the *soundness-preserving* criteria used in this paper are defined with respect to traditional monolithic lifting that performs whole program analysis, i.e., the result of polylithic lifting should recover as much information as the result of the monolithic one. More formally, the *soundness-preserving* criteria concern three important code properties: control-flows, data-flows, and types, and are defined as follows.

Definition 2.1. (*soundness-preserving*) Given an input binary B, suppose the monolithic lifter produces the IR $\mathcal{I}_{mono}$, and DIATOM produces $\mathcal{I}_{poly}$. $\mathcal{I}_{poly}$ is soundness-preserving with respect to $\mathcal{I}_{mono}$ if:

- (Control-flow soundness) Given $\mathcal{I}_{mono}$'s CFG $G = (V, E)$, the CFG for $\mathcal{I}_{poly}$ $G' = (V', E')$ is control-flow sound iff $V \subseteq V' \wedge E \subseteq E'$.
- (Data-flow soundness) Let D and D' be the set of high-level variables, parameters, or return values recovered by $\mathcal{I}_{mono}$ and $\mathcal{I}_{poly}$. $\mathcal{I}_{poly}$ is data-flow sound with respect to $\mathcal{I}_{mono}$ iff $D \subseteq D'$.
- (Type soundness) Let v be any value inside D, and v' be the corresponding value inside D'. $\mathcal{I}_{poly}$ is type sound iff the type of v' can be safely used by v without breaking the program.

Algorithm 1 shows the polylithic parallel binary lifting algorithm. Similar to conventional binary lifters, it takes a single binary program as the input and produces the translated LLVM IR through multiple stages. DIATOM realizes the polylithic design by splitting the monolithic LLVM module into smaller pieces to perform parallel binary lifting. By breaking the single module into multiple ones, translating each partition can be totally isolated from others and thus can be fully parallelized. In the meantime, it leverages light-weight data-flow summaries and type-aware IR linking to avoid soundness loss caused by separating dependent code fragments. Specifically, DIATOM mainly consists of five phases (more implementation details are in § 7).

Phase 1. In the first phase, the input binary is disassembled to produce the monolithic IR module $\mathcal{I}_{dis}$ (Line 2) using a combination of recursive descent and linear sweep algorithms [80], similar to existing binary lifters [13, 34, 37, 38, 57, 92]. Disassembly contains two sub-tasks: instruction lifting and inter-procedural control-flow graph (ICFG) construction. These two tasks are intertwined [31]: assembly instructions are translated into LLVM IRs by making their side-effect explicit [53], and in the meantime, the produced LLVM statements are analyzed to find more instruction addresses and build the ICFG. For example, data-flow analysis is used to identify targets of indirect control-flow transfers [75]. After disassembly, the original machine code is translated into LLVM IRs, which eliminates the need for reasoning about complex assembly language. In the rest of the paper, all the proposed algorithms are discussed based on IR languages. Moreover, all function boundaries will be determined and will not be changed anymore, which enables function-level parallelism. Since the disassembly step is performed monolithically, the control-flow soundness is naturally guaranteed, i.e., it produces the same result as the monolithic lifting.

Phase 2. When the initial LLVM IR is generated, DIATOM computes light-weight data-flow summaries for all functions, which are then used by the parallel threads as the global information for maintaining the data-flow soundness (Line 3). The summary generation is efficient since it only tracks non-aliasing global registers, and the generation process can be fully parallelized.

Phase 3. Then, DIATOM splits all functions inside $\mathcal{I}_{dis}$ into n parts, where n is the number of available threads (Line 5 - Line 7).

Phase 4. After IR splitting, DIATOM continues to perform other lifting steps, such as variable recovery and code optimizations on the split IRs. Different from monolithic lifting that performs

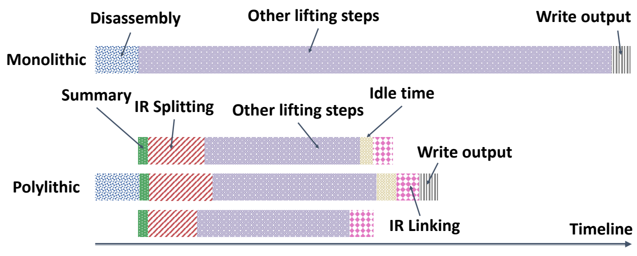


Fig. 3. Execution timeline for the two different lifter designs with three threads.

these steps with a single thread, Diatom runs these steps on all IR partitions in parallel asynchronously (Line 8). The data-flow summaries are used to prevent soundness loss caused by separating dependent functions.

Phase 5. Finally, IR partitions are merged to produce the final output $\mathcal{I}_{link}$ with a type-aware IR linking process (Line 9). During linking, conflict types for the same code entity shared by different partitions are resolved by exploiting subtyping relations to maintain the type soundness.

Figure 3 shows a comparison of the execution timeline between the traditional monolithic design and Diatom's new polylithic design using 3 threads. As shown in the figure, the monolithic design can only utilize a single thread and thus suffers from scalability issues. Diatom runs the summary generation (phase 2), the IR splitting (phase 3), and other lifting steps (phase 4) in a fully parallelized way. The linking step (phase 5) is performed in serial in an asynchronous way with locking, thus producing short idle time.

In the following sections, we will first give some preliminary definitions (§ 3), and then we will discuss in detail the three main components of Diatom that are different from traditional binary lifters: the data-flow summary generation (§ 4), the IR splitting (§ 5), and the type-aware IR linking (§ 6). More implementation details of other lifting steps can be found in § 7.

3 Preliminary Definitions

Language. To present our work formally, we use a simple LLVM-like static single assignment form (SSA) call-by-value language shown in Figure 4 to model the target program, similar to the previous work [94, 95]. A program is in the static single assignment form [30] and may have multiple functions, with or without a function body. The semantics of statements are standard. Statements in this language include common assignments, ϕ-assignments, binary and unary operations, loads, stores, branches, returns, calls, and sequencing. As discussed in § 2, the translation from original binary instructions to the IR language is performed during the disassembly step, which is not related to the contribution of this paper. Other lifting steps are all based on the IR language presented in Figure 4, which abstracts away lots of machine-specific features. Registers are declared as integer-typed global variables, and stack memory is accessed through the stack pointer register.

Type system. Figure 4 also shows the syntax for types. To handle types in different LLVM module environments, Diatom adopts a nominal type system [70, 99] (similar to the one used in C/C++), in which types are identified by names instead of definitions so that the equivalence among types can be established on the basis of their names. It contains primitive types such as $\texttt{int}_{sz}$, `double`, and `float`. $\texttt{int}_{sz}$ is an integer type with bit size sz. For example, in a 64-bit `X86_64` system, the type of general register is $\texttt{int}_{64}$. $\tau*$ represents the pointer type, with τ as the element type. $\tau[n]$ is an array type of size n. A class or structure type is labeled with a name x, and a sequence of field types $\{\tau_1, \tau_2, ...\}$. Although LLVM's type system does not have separate notions for class and structure types, and uses the `StructType` to represent both of types, at the LLVM IR level, these two kinds

$$\begin{array}{rl}
\text{Program} & P := F+ \\
\text{Function} & F := f(v_1, v_2, \ldots) = \{S;\} \\
 & \quad \mid f(v_1, v_2, \ldots) = \varnothing \\
\text{Statement} & S := v_1 = \&v_2 \mid v_1 = v_2 \mid v_1 = *v_2 \\
 & \quad \mid *v_1 = v_2 \mid v_1 = v_2 \oplus v_3 \\
 & \quad \mid v = \phi((\varphi_1, v_1), (\varphi_2, v_2), \ldots) \\
 & \quad \mid r = \textbf{call } f(v_1, v_2, \ldots) \mid r = \textbf{call } v(v_1, v_2, \ldots) \\
 & \quad \mid \textbf{return } v \mid \textbf{if}(v)\,\{S_1;\}\ \textbf{else}\,\{S_2;\} \mid S_1; S_2 \\
 & \oplus \in \{\wedge, \vee, +, -, >, <, \equiv, \neq, \ldots\}
\end{array}$$

$$\begin{array}{rl}
\text{Type} & \tau := \mathtt{int}_{sz} \\
 & \quad \mid \mathtt{double} \mid \mathtt{float} \\
 & \quad \mid \mathtt{void} \\
 & \quad \mid \tau* \\
 & \quad \mid \tau[n] \\
 & \quad \mid (\tau_1, \tau_2, \ldots) \rightarrow \tau \\
 & \quad \mid \mathtt{struct}\ x\,\{\tau_1, \tau_2, \ldots\} \\
 & \quad \mid \mathtt{class}\ x\,\{\tau_1, \tau_2, \ldots\}
\end{array}$$

Fig. 4. The syntax and the types of the language.

of types usually have different prefixes in their names, i.e., "class." and "struct.". Therefore, we can distinguish them based on the names. During lifting, it is also possible to distinguish class and structure types by analyzing their semantics, for example, there could be dynamic dispatch/virtual methods for class-typed data. A function type, $(\tau+) \rightarrow \tau$, consists of a tuple that contains the types of a function's formal parameters mapped to the type of a function's return.

Abstract domains. Given a program P, a label $\ell \in L$ indicates the position of a statement in the control-flow graph of the IR. Following previous work [44, 127], we abstract the low-level machine code as a collection of registers $r \in \mathcal{R}$, which is globally shared by all the code for passing data between instructions. The whole memory space is split into disjoint regions: global, stack, and heap. Abstract values are represented as $\langle m, o, sz \rangle$, where m stands for a memory region and o stands for the offset relative to the base of the region. The global region, denoted as $\mathcal{M}_g$, stands for the locations holding initialized and uninitialized global data, such as the `.data` and `.rodata` segments of an `ELF`. A stack region, denoted as $\mathcal{M}_s^f$, represents the stack frame for function f and holds local variable values. $\mathcal{M}_h^\ell$ denotes a heap region that is allocated at $\ell \in L$.

4 Data-flow Summary Generation

Since functions are distributed based on their sizes, functions with inter-procedural dependences (e.g., caller-callee relations) could be placed into different partitions. However, several lifting stages require reasoning about inter-procedural data-flows between functions, such as parameter passing [92] and stack memory sharing [92, 130]. We discuss more implementation details in § 7. Unaware of the inter-procedural data-flow information during lifting could threaten the data-flow soundness of lifted IRs, as illustrated in § 1.3.

One crucial task of a binary lifter is to abstract away from the low-level data-flows implemented with machine registers and stack memory, and transform them into high-level ones, which use function parameters to pass data between procedures. The above task requires data-flow analysis on both caller and callee functions. The low-level machine code leverages two machine features to implement inter-procedural data-flows between functions: global registers and stack memory [38]. Global registers are usually written by the caller function and read by the callee function to pass necessary parameters. Each function maintains its own stack frame, which is a range of stack memory that can be manipulated through the stack pointer register [64]. The stack frame accommodates both locally declared variables and actual parameters. The stack space for parameters is shared with the stack frame of the caller function.

Algorithm 2: Summary Generation

```
Input: Translated LLVM IR after Disassembly $\mathcal{I}_{dis}$
Output :Generated function summary map $\psi_{\mathcal{S}}$
1  Function ComputeSummary($\mathcal{I}_{dis}$):
2      $\mathcal{G} \leftarrow$ getCallGraph($\mathcal{I}_{dis}$)                          ▷ generate the program call graph
3      $\psi_d \leftarrow []$                                                  ▷ function data-flow facts mapping
4      foreach $f$ in $\mathcal{G}$ do in parallel
5          $\mathcal{S}_i^f, \mathcal{S}_o^f \leftarrow \varnothing$                                     ▷ input/output summary
6          $\mathbb{R}, \mathbb{I}, \mathbb{O} \leftarrow$ getDataflowFacts($f$)                     ▷ forward data-flow tracking
7          $\psi_d[f \mapsto (\mathbb{R}, \mathbb{I}, \mathbb{O})]$                         ▷ record the results of data-flow tracking
8          $\psi_{\mathcal{S}}[f \mapsto (\mathcal{S}_i^f, \mathcal{S}_o^f)]$                                  ▷ update summary map
9      $W \leftarrow \varnothing$                                                          ▷ empty worklist
10     foreach $f$ in $\mathcal{G}$ do
11         $W \leftarrow W \cup f$
12     while $W$ is not empty do
13         $f \leftarrow$ pop from $W$
14         $\mathbb{R}, \mathbb{I}, \mathbb{O} \leftarrow \psi_d(f)$                                      ▷ get data-flow facts of f
15         $\mathcal{S}_i^f, \mathcal{S}_o^f \leftarrow \psi_{\mathcal{S}}(f)$                              ▷ already computed summaries for f
16         foreach $\mathcal{C}$ in getCallee($f, \mathcal{G}$) do                   ▷ analyze each call site inside f
17             $c \leftarrow$ get the called function at $\mathcal{C}$                          ▷ c is the called function
18             $\mathcal{S}_i^c, \mathcal{S}_o^c \leftarrow \Pi_{\mathcal{S}}(c)$                               ▷ already computed summaries of c
19             foreach $i$ in $\mathcal{S}_i^c$ do
20                 if $\Pi_{\mathbb{R}}(i, \ell_{\mathcal{C}}) = \varnothing$ then
21                     $\mathcal{S}_i^f \leftarrow \mathcal{S}_i^f \cup i$                           ▷ add new values to the input summary
22             foreach $o$ in $\mathcal{S}_o^c$ do
23                 $\mathbb{R}, \mathbb{I}, \mathbb{O} \leftarrow$ updateDataflowFacts($\mathcal{C}$)                ▷ update data-flow facts
24         foreach $i$ in $\mathcal{S}_i^f$ do
25             if $\Pi_{\mathbb{R}}(i, \ell) \neq \varnothing$ then
26                 $\mathcal{S}_i^f \leftarrow \mathcal{S}_i^f \backslash i$                  ▷ remove defined values from the output summary
27         if $(\mathcal{S}_i^f, \mathcal{S}_o^f)$ changes then
28             foreach caller in getCaller($f, \mathcal{G}$) do
29                 $W \leftarrow W \cup$ caller )                                     ▷ add caller of f to the worklist
30     return $\psi_{\mathcal{S}}$
```

Example 4.1. The r8d register is defined at Line 13 and read at Line 4 in Figure 2b ($r8d_{foo} \rightsquigarrow r8d_{bar}$), which will be identified as a function parameter. The stack memory accesses at Line 11 and Line 6 in Figure 2b refer to the same stack location, which is defined in foo and read in bar ($[rsp]_{foo} \rightsquigarrow [rsp + 8]_{bar}$), therefore will be marked as a stack parameter.

If functions with caller and callee relations are separated into two partitions, the inter-procedural data-flow relations between registers and stack memory cannot be established, which could result in unsound data-flow recovery results, i.e., missing data-flows.

Example 4.2. In Figure 2b, if `foo` and `bar` are separated into two partitions, the lifter will not be aware of use sites for the stack memory write at Line 11 inside the callee function `bar` ($[rsp]_{foo} \rightsquigarrow \varnothing$). Therefore, the stack parameter is missing, causing incorrect lifting results.

Our key idea for solving this problem is to compute summaries for each function before IR partitioning so that the caller function can be aware of the callee's data-flow facts even in the absence of its function body. The data-flow summary is meant to be general; it can be used to facilitate any lifting stages that require the callee information.

Since binary lifting only aims to abstract away low-level machine operations through global registers and stack memory [28], we only need to track global registers that cannot be address-taken; therefore, we do not need to consider possible aliasing relations on heap memory during data-flow recovery, making the analysis light-weight. Moreover, the summary computation is performed intra-procedurally and does not modify the original IR so that the process can be parallelized based on functions. Experimental results also show that the analysis can scale to extremely large binaries; on average, the summary computation only accounts for 3.2% of the total running time in Diatom. The data-flow summary is defined as follows.

Definition 4.1. The data-flow summary Δ for function f is a pair $\Delta = (\mathcal{S}_i^f, \mathcal{S}_o^f)$, which denotes f's input and output summaries, respectively. $\mathcal{S}_i^f$ is the input summary of f, which is a set of registers and stack memory slots that are defined by f's caller function. $\mathcal{S}_o^f$ is the output summary of f, which contains the set of registers that are defined inside f and will be returned to its caller.

Example 4.3. For the function `bar` in Figure 2b, its data-flow summary is computed as $\Delta = (\mathcal{S}_i^{bar}, \mathcal{S}_o^{bar})$, where $\mathcal{S}_i^{bar} = \{esi, edx, ecx, r8d, r9d, [rsp+8], rdi\}$, and $\mathcal{S}_o^{bar} = \varnothing$. `rdi` is inside $\mathcal{S}_i^{bar}$ because at Line 7, the value of `rdi` is loaded but `rdi` is not defined inside `bar`.

We also would like to note that the generated data-flow summaries have already captured all required information to handle complex parameters/return values passing mechanisms in binary code. For example, programs can pass arrays/structures by value using function parameters. At the LLVM IR-level, passing those aggregate values as parameters is often implemented using pointer-based or memory-based mechanisms. When lowering to binary code, those values or pointers are still passed through registers or stack memory reference, which are already captured by our dataflow summaries. It is also common for programs to pass heap-allocated pointers to functions as parameters and also take the return values. At the binary level, such pointers are still passed through either global registers or stack memory, and both of them are captured by summaries.

4.1 Summary Generation Algorithm

The summary generation process performs forward traversal on the function body to track data-flows of the stack pointer and registers for parameter passing. As shown in Algorithm 2, for each function f inside the call graph $\mathcal{G}$, Diatom computes both input summary $\mathcal{S}_i^f$ and output summary $\mathcal{S}_o^f$. The summary contains two kinds of code entities that could propagate values inter-procedurally in binary code: registers and stack memory [38].

Figure 5 formulates the intra-procedural data-flow tracking rules for basic statements that are used by the `getDataflowFacts()` procedure of the algorithm (Line 3 - Line 8). Each rule is in the form $\mathbb{R}, \mathbb{I}, \mathbb{O} \vdash \ell : stmt : \mathbb{R}', \mathbb{I}', \mathbb{O}'$, which states that given the current register state map $\mathbb{R}$, the input map $\mathbb{I}$, and the set of output value $\mathbb{O}$, the statement $stmt$ at program point ℓ updates them into $\mathbb{R}'$, $\mathbb{I}'$, and $\mathbb{O}'$. The register map $\mathbb{R}(r) = \{(\ell, v)\}$ denotes that the register r has the value v at program point ℓ. The input map $\mathbb{I}(v) = \{\ell\}$ denotes that the value v is defined by the caller function and is used at program point ℓ in the current function. The output set $\mathbb{O} = \{v\}$ denotes that v's definition

$$\frac{}{(\mathbb{R}_0, \mathbb{I}_0, \mathbb{O}_0) \vdash \ell_0 : \mathsf{entry}(f) : (\mathbb{R}_0[r_{sp} \mapsto (\ell_0, 0)], \mathbb{I}_0, \mathbb{O}_0)} \quad (\textsc{InitState})$$

$$\frac{\mathbb{O}' = \begin{cases} \{r \mid r \in \mathbb{R}\} & \text{if } \mathbb{O} = \varnothing \\ \{r \mid r \in \mathbb{R} \wedge r \in \mathbb{O}\} & \text{otherwise} \end{cases}}{(\mathbb{R}, \mathbb{I}, \mathbb{O}) \vdash \ell : \mathsf{ret} : (\mathbb{R}, \mathbb{I}, \mathbb{O}')} \quad (\textsc{Ret})$$

$$\frac{\mathbb{R}' = \mathbb{R}[r \mapsto (\ell, q) \cup \mathbb{R}(r)]}{(\mathbb{R}, \mathbb{I}, \mathbb{O}) \vdash \ell : r := q : (\mathbb{R}', \mathbb{I}, \mathbb{O})} \quad (\textsc{WriteReg}) \qquad \frac{\Pi_{\mathbb{R}}(r, \ell) = \varnothing \Rightarrow \mathbb{I}' = \mathbb{I}[r \mapsto \ell \cup \mathbb{I}(r)]}{(\mathbb{R}, \mathbb{I}, \mathbb{O}) \vdash \ell : p := r : (\mathbb{R}, \mathbb{I}', \mathbb{O})} \quad (\textsc{ReadReg})$$

$$\frac{\begin{array}{c}\mathbf{getMem}(q) \in \mathcal{M}_s^f \quad o = \mathbf{getOffset}(q) > 0 \Rightarrow \mathbb{I}' = \mathbb{I}[\left\langle \mathcal{M}_s^f, o, \mathbf{sizeof}(p) \right\rangle \mapsto \ell] \\ \mathbb{R}' = \begin{cases} \mathbb{R}[p \mapsto (\ell, *q)] & \text{if } p \in \mathbb{R} \\ \mathbb{R} & \text{otherwise} \end{cases}\end{array}}{(\mathbb{R}, \mathbb{I}, \mathbb{O}) \vdash \ell : p := *q : (\mathbb{R}', \mathbb{I}', \mathbb{O})} \quad (\textsc{ReadMem})$$

$$\frac{\mathbb{R}, \mathbb{I}, \mathbb{O} \vdash \mathcal{S}_1 : \mathbb{R}_1, \mathbb{I}_1, \mathbb{O} \quad \mathbb{R}, \mathbb{I}, \mathbb{O} \vdash \mathcal{S}_2 : \mathbb{R}_2, \mathbb{I}_2, \mathbb{O} \quad \mathbb{R}' = \mathbb{R}[r \mapsto (\ell, \gamma) \mid \exists \ell_1 \in \mathbb{R}_1 \wedge \ell_2 \in \mathbb{R}_2]}{(\mathbb{R}, \mathbb{I}, \mathbb{O}) \vdash \ell : \mathbf{if}(v)\ \{\mathcal{S}_1\}\ \mathbf{else}\ \{\mathcal{S}_2\} : (\mathbb{R}', \mathbb{I}_1 \cup \mathbb{I}_2, \mathbb{O})} \quad (\textsc{Branching})$$

$$\frac{\mathbb{R}, \mathbb{I}, \mathbb{O} \vdash \mathcal{S}_1 : \mathbb{R}', \mathbb{I}', \mathbb{O}' \quad \mathbb{R}', \mathbb{I}', \mathbb{O}' \vdash \mathcal{S}_2 : \mathbb{R}'', \mathbb{I}'', \mathbb{O}''}{(\mathbb{R}, \mathbb{I}, \mathbb{O}) \vdash \mathcal{S}_1; \mathcal{S}_2 : (\mathbb{R}'', \mathbb{I}'', \mathbb{O}'')} \quad (\textsc{Sequencing})$$

Fig. 5. Intra-procedural data-flow tracking rules.

is available after the current function returns. We also define the operation $\Pi_{\mathbb{R}}(r, \ell)$ for querying the definition for register r before the program point ℓ. Specifically, we handle read/write from/to registers and memory. We also handle program constructs such as sequencing and branching.

- We analyze each function independently of its calling context. The rule InitState sets up the initial state for the function f. It initializes the stack pointer register to be 0 to represent the bottom of the stack frame at function start (ℓ_0). Other registers are assumed to be uninitialized at the function entry and, therefore, do not have any mappings in $\mathbb{R}$.
- The rule Ret handles the return statement. It takes the intersection of all registers that are defined before each return statement.
- The rule WriteReg is used for updating the register state map $\mathbb{R}$, which binds the value q to the register r at program point ℓ.
- The rule ReadReg checks whether $\mathbb{R}$ contains r, i.e., whether r has been defined by other statements before ℓ. If not, it will add r to $\mathbb{I}$, indicating that the loaded value of r at ℓ is possibly defined by f's caller.
- The rule ReadMem handles memory read such as the `mov [rsp],4` instruction in Figure 2b. It uses procedures **getMem** and **getOffset** to check whether the instruction refers to stack memory $\mathcal{M}_s^f$ at a certain offset o. The two procedures can be implemented effectively by analyzing operations on the stack pointer (e.g., `RSP` in `X86_64`) without expensive pointer analysis [1, 6, 7, 19, 43, 130]. Specifically, in order to uniformly represent stack offsets, for each instruction that accesses the memory, DIATOM performs a backward data-flow analysis to obtain the current stack offset relative to the function start (o). If o is outside of the stack

frame for f, i.e., the offset is larger than 0, the accessed memory $\left\langle \mathcal{M}_s^f, o, \textbf{sizeof}(p) \right\rangle$ will be added to $\mathbb{I}$. The auxiliary function **sizeof** retrieves the bit size of the memory read/write. Writing to memory is handled similarly.
- The rule Branching handles multiple paths. It updates $\mathbb{R}(r)$ by merging multiple definitions into an abstract value γ if r is defined on all paths. Otherwise, it will preserve the information that r is still undefined on some paths. It also merges the input map $\mathbb{I}$ from all paths.
- The rule Sequencing is simple, which uses the post state of statement $\mathcal{S}_1$ to analyze the immediate next statement $\mathcal{S}_2$.

Summary propagation. However, only analyzing the function body could result in missing or fake inter-procedural data-flows since the above rules do not handle call site statements. An intuitive example is the function wrappers [103], which may not have explicit definitions for parameters before call sites.

Example 4.4. In Figure 2b, the three parameters of `foo` are directly used by the call site of `bar` at Line 15. Therefore, no explicit `mov` instructions are used for parameter registers `rdi`, `rsi` and `rdx`. Therefore, merely analyzing `foo`'s function body could miss data-flows.

Example 4.5. Consider a simple `X86_64` assembly function $f = \{call\ f_1;\ load\ xmm0;\}$. Merely analyzing the function body of f could conclude that the input summary $\mathcal{S}_i^f$ contains register $xmm0$. However, $xmm0$ is also used to store floating-point return values. Therefore, if f_1 is returning a floating-point value, the $xmm0$ inside f is actually implicitly defined by $call\ f_1$ and should not be added to $\mathcal{S}_i^f$, which could cause fake data-flows.

To tackle this problem, Line 9 - Line 29 of the algorithm further propagates local function summaries through the call graph until a fixed point is reached. For a function f inside the call graph $\mathcal{G}$, the algorithm iterates through all call sites inside f's function body (Line 17). For each callee function c, the algorithm checks whether input values inside $\mathcal{S}_i^c$ have definitions in the current function by performing a query $\Pi_{\mathbb{R}}$ on the current function, if not, the input value i is further propagated to f (Line 21). Using the output summary $\mathcal{S}_o^c$ of c, the algorithm updates the data-flow facts of f (Line 23). The input summary of $\mathcal{S}_i^f$ is further refined by removing values that are defined by the updated $\mathbb{R}$ (Line 26). If f's data-flow summaries are updated, we add all f's caller functions to the worklist (Line 29), until a fixed-point is reached.

Loop handling. Algorithm 2 is an intra-procedural flow- and path-insensitive analysis, with a propagation on the direct call graph. It computes input/output values of functions regarding registers and stack references. For registers, since we only want to know whether they are defined within the function body, loops can be unrolled once to reason about the def-use relations. For stack memory, our algorithm is mainly concerned with the base offset of each accessed stack memory. For example, if there is an array on the stack, the memory could be accessed with $[rsp + 10 + index * 4]$. The algorithm mainly extracts the relative offset to the stack frame for the base pointer, which is $rsp + 10$; therefore, loops also will not affect the analysis.

Soundness of summaries. The following theorems explain why the summary generation preserves the data-flow soundness in Definition 2.1.

Theorem 4.1 (Summary soundness). At the fixed point of Algorithm 2, for any function f and any tracked value v (non-aliasing registers or stack slots), if v is used in f and no intra-procedural definition reaches that use, then $v \in \mathcal{S}_i^f$. Moreover, for any register v, if v is defined on all paths to every return in f, then $v \in \mathcal{S}_o^f$.

Proof sketch. The intra-procedural rules in Figure 5 add each use without a prior local definition to $\mathbb{I}$, and the propagation phase adds the callee inputs that remain undefined at each call site to the caller's $\mathcal{S}_i^f$. The worklist reaches a fixed point on the call graph, so all inter-procedural uses are captured. The RET rule intersects register definitions at returns, yielding exactly the registers defined on every return path in $\mathcal{S}_o^f$.

Theorem 4.2 (Summary-based data-flow soundness). Let $\mathcal{I}_{mono}$ be the IR produced by monolithic lifting and $\mathcal{I}_{poly}$ be the IR produced by polylithic lifting that uses $\psi_{\mathcal{S}}$ (the function summary map) for parameter/return recovery, under the same register/stack abstraction and calling convention. Let D and D' be the sets of recovered parameters and return values in $\mathcal{I}_{mono}$ and $\mathcal{I}_{poly}$, respectively. Then $D \subseteq D'$.

Proof sketch. Any parameter/return recovered by the monolithic lifter corresponds to a caller-defined register/stack slot or a callee return register observable in the full program. By Theorem 4.1, these values appear in $\mathcal{S}_i^f$ or $\mathcal{S}_o^f$, so the parallel lifting steps that use $\psi_{\mathcal{S}}$ reconstruct at least the same parameters/returns in each partition. Linking does not remove these entities; therefore, $D \subseteq D'$.

5 IR Splitting

One of the key steps that enable DIATOM's polylithic design is IR splitting, which decomposes the monolithic LLVM module into individual partitions to parallelize the binary lifting process. A naive way for splitting the IR into independent partitions is to use the `SplitModule` functionality in the LLVM framework [8, 50, 112]. It first copies the original IR into a serial of new IRs $\{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_n\}$. Then, it serializes new IRs to disk files and finally unserializes files into modules with individual environments $\{\mathcal{E}_1, \mathcal{E}_2, ..., \mathcal{E}_n\}$. However, the naive approach only performs a "shallow copy", which only copies IR statements syntactically that keep the references to the original IR, meaning that all cloned IR modules still share the same global environment $\mathcal{E}$ as the original one. Therefore, the splitting process has to be performed in serial with a single thread, which causes a severe slowdown and even shadows the performance gain brought by parallelism. In our preliminary evaluation, we found that the naive approach could even take over one hour to split the IR of large binaries.

To overcome the above limitations, our key idea is to perform a "deep copy" during IR splitting so that new IR modules are already independent in terms of global environments. The idea provides a "one-stone-two-birds" solution: it enables a fully parallelized splitting process while eliminating the redundant serialization-deserialization overhead.

Function Distribution. The `DistributeFunctions` procedure in Algorithm 1 is used for assigning functions inside $\mathcal{I}_{dis}$ into n groups, where n is the number of available threads. DIATOM's function distribution strategy is based on a key observation that the lifting time for specific functions can be approximated by their sizes. Therefore, our strategy aims to evenly distribute functions to different partitions according to their sizes. We formulate the function distribution problem as a variant of the classic k-partition problem [18]: *given a set of n functions $\{f_0, f_1, ..., f_n\}$ where f_i is of size s_i (count of instructions), find a partition $\{S_0, S_1, ..., S_k\}$ of size k (the number of available threads) such that the difference between the subset sums is minimized.*

The partition problem is $\mathcal{NP}$-hard, and we use the popular greedy algorithm to solve it [48]. Specifically, it computes a mapping $\psi_d : f \mapsto i$ to indicate the partition i where f should belong. The high-level algorithm iterates through all the function sizes in descending order, assigning each of them to whichever subset has the smaller sum. The running time is only $O(n\log n)$; therefore, it can scale to extremely large binaries.

Lock-free IR Cloning. The `Clone` procedure in Algorithm 1 copies functions inside $\mathcal{I}_{dis}$ into one partition $\mathcal{I}_i$. The major advantage of this algorithm is two-fold. First, it does not involve any lock mechanism since $\mathcal{I}_i$ has an independent global environment $\mathcal{E}_i$; therefore, it can be fully parallelized. Second, the cloning is performed in a pure on-demand manner, i.e., values are only cloned when instructions refer to them. Specifically, to clone the function f_o from $\mathcal{I}_{dis}$ into f_n inside

$\mathcal{I}_i$, it maintains three mappings between $\mathcal{I}_{dis}$ and $\mathcal{I}_i$: ψ_g maps global values such as functions and constants, ψ_t maps types, and ψ_l maps local values between f_o and f_n. Then, it performs a "deep copy" on f_o, which rebuilds all statements from scratch, including all the referred values such as constants, global variables, and functions. For functions that do not belong to the current group, only declarations are created instead of full-function bodies.

6 Type-aware IR Linking

Algorithm 3: Type-aware IR Linking

```
   Input: Translated LLVM IRs $\mathcal{I}_i, \mathcal{I}_j$
   Output : Linked LLVM IR $\mathcal{I}_{link}$
1  Function Link($\mathcal{I}_i, \mathcal{I}_j$):
2      foreach global values $v_i$ in $\mathcal{I}_i$ do
3          if $\mathcal{I}_j$ contains the same code entity $v_j$ then
4              $\tau_i \leftarrow$ getType($v_i$), $\tau_j \leftarrow$ getType($v_j$)    ▷ retrieve the type of the value
5              if $\tau_i \preceq \tau_j$ then
6                  modify $v_j$ to $\tau_i$ inside $\mathcal{I}_j$    ▷ modify the type and all use sites
7              else if $\tau_j \preceq \tau_i$ then
8                  modify $v_i$ to $\tau_j$ inside $\mathcal{I}_i$
9              else
10                 $\tau_m \leftarrow$ Merge($\tau_i, \tau_j$)    ▷ merge two types
11                 modify $v_i, v_j$ to $\tau_m$ inside $\mathcal{I}_i, \mathcal{I}_j$
12     $\mathcal{I}_{link} \leftarrow$ Link($\mathcal{I}_i, \mathcal{I}_j$)
```

After DIATOM finishes lifting IR partitions, it links IRs together to form the final output $\mathcal{I}_{link}$. Since all partitions are merged into a single LLVM module, the linking step has to be performed in serial, and DIATOM uses the lock mechanism to avoid possible data races.

The main challenge of the linking step is that types of shared global values across IR partitions could be recovered differently, which could break the type soundness during IR linking. Global values such as functions or global variables can be referred to by instructions in different IR partitions. For example, different functions could refer to the same global memory address in the function body, which is defined as follows.

Definition 6.1. Global value v_i inside IR $\mathcal{I}_i$ refers to the same code entity as v_j inside $\mathcal{I}_j$ if v_i and v_j have the same **symbol name**.

Existing binary lifters [37, 57, 92, 130] incorporate simple type inference mechanisms into the lifting process. For example, LASAGNE [92] proposes to promote integer-typed parameters of a given function to a pointer type by collecting and analyzing all their uses. However, since each partition only contains a partial set of all the uses for global values, the type for the same value might be recovered differently across IR partitions.

Example 6.1. In Figure 2, the first parameter of the function `bar` (`rdi`) can be inferred as a pointer type `int`$_{64}$`*` according to the memory read operation at Line 7. Therefore, the type of the first callee parameter at Line 15 will also be declared as a pointer type. However, if `foo` and `bar` are split into two individual partitions, the callee parameters of `bar` will be different; thus, these two IRs cannot be directly linked together and could break the type soundness.

$$\frac{}{\tau \leq \tau}\ (\textsc{Reflexivity}) \qquad \frac{\tau \leq \tau' \quad \tau' \leq \tau''}{\tau \leq \tau''}\ (\textsc{Transitivity}) \qquad \frac{}{\tau \leq \texttt{void}}\ (\textsc{Void})$$

$$\frac{\mathbf{sizeof}(\tau) \leq \mathbf{sizeof}(\tau')}{\tau \leq \tau'}\ (\textsc{Primitive}) \qquad \frac{\tau \leq \tau'}{\tau{*} \leq \tau'{*}}\ (\textsc{Pointer}) \qquad \frac{\tau \leq \tau' \quad n \geq m}{\tau[n] \leq \tau'[m]}\ (\textsc{Array})$$

$$\frac{[\tau_i \leq \tau_i']_{i \in 0,\dots,m} \quad \tau \leq \tau' \quad n \geq m}{(\tau_1, \tau_2, \dots, \tau_n) \to \tau \leq (\tau_1', \tau_2', \dots, \tau_m') \to \tau'}\ (\textsc{Function}) \qquad \frac{[\tau_i \leq \tau_i']_{i \in 0,\dots,m} \quad n \geq m}{x\,\{\tau_1, \tau_2, \dots, \tau_n\} \leq x'\,\{\tau_1', \tau_2', \dots, \tau_m'\}}\ (\textsc{Aggregate})$$

$$\frac{\tau_1 \leq \tau}{x\,\{\tau_1, \tau_2, \dots, \tau_n\} \leq \tau}\ (\textsc{FirstMember}) \qquad \frac{\tau' \in \texttt{int}_{sz} \ \mathbf{sizeof}(\tau) \leq \mathbf{sizeof}(\tau')}{\tau{*} \leq \tau'}\ (\textsc{PointerInteger})$$

Fig. 6. Inference rules that define the subtyping relations.

Our key idea for solving the above challenge is to leverage the subtyping relations to reconcile type discrepancies between IRs while maintaining soundness. We propose a type-aware IR linking algorithm to reconcile shared code entities among IR partitions, as shown in Algorithm 3. It works by examining all shared global values (e.g., functions and global variables) between IRs. For the two shared global values v_i and v_j inside $\mathcal{I}_i$ and $\mathcal{I}_j$, if their types form subtyping relations, the algorithm always downcast [97, 100] one of them to another, otherwise, their types are merged.

6.1 Subtyping Relations

Subtyping [25, 83, 87, 97, 118] enables safe substitution between different types and is often considered an essential feature of object-oriented languages. We formalize the intuition that some types are more informative (precise) than others but still have a compatible runtime representation. Specifically, if τ is a subtype of τ', written $\tau \leq \tau'$, it means that any term of type τ can safely be used in a context where a term of type τ' is expected. The subtyping rules, which are mostly standard [87, 121], are shown in Figure 6. In the figure, most rules follow the original definitions of physical-subtyping [25], and there are several additional rules specialized for machine code, similar to the one proposed by Xu et al. [118].

REFLEXIVITY and TRANSITIVITY are two basic rules indicating that any type τ is a subtype of itself, and if τ is a subtype of τ' and τ' is a subtype of τ'', then τ is also a subtype of τ''.

Besides basic subtyping rules, we consider each form of the type defined in Figure 4 (function types, structure types, etc.); for each one, we introduce one or more rules formalizing situations when it is safe to allow elements of one type of this form to be used where another is expected.

- The rule VOID indicates that any type τ can be a subtype of the `void` type, i.e., a τ type can safely be used in a context where a `void` type is expected.
- Rule PRIMITIVE states that a primitive type τ is a subtype of type τ' if τ' has at least as many bits as τ. For example, the usage of a 32-bit integer can be safely replaced by an 8-bit integer.
- The rule POINTER states if τ is a subtype of τ', then $\tau{*}$ is a subtype of $\tau'{*}$. For example, rules VOID and POINTER reflect that `void*` is the supertype of other pointer types.
- The rule ARRAY indicates that an array $\tau[n]$ is a subtype of $\tau'[m]$ if $\tau \leq \tau'$ and $\tau[n]$ is longer.
- The rule FUNCTION defines a subtyping relation between two function types T_1 and T_2. T_1 can safely substitute T_2 if T_1 has more parameters, its return type is a subtype of T_2's return type, and each parameter type is also a subtype of T_2's.

- The rule Aggregate is defined in a similar way as the rule Function, it defines the subtyping relation between two aggregate types (structure and class types). Note that LLVM does not define a first-class union type in its IR, instead, a C union is lowered into a structure type whose size and alignment match the largest and most strictly aligned union member. Inside IR statements, a particular member of the original union type is accessed through a pointer cast (via **bitcast**). Therefore, when we merge shared global values that have union types, we can still apply the Aggregate rule to obtain the linked type.
- The rule FirstMember defines that a structure is a subtype of another type τ if the type of the first member τ_1 is a subtype of τ, i.e., it is safe to use a structure in a place where a supertype of its first member is expected.
- The rule PointerInteger indicates that a pointer type could be the subtype of an integer type. All values inside the primitive binary code do not have explicit type declarations and are operated as if they are integers through global registers or memory. Therefore, if a binary-level value is inferred as a pointer, it can still be safely used in the original binary code where an integer is expected. For example, the `X86_64` **add** instruction can be used to add two integers, but also to add an integer and a pointer.

6.2 IR Linking Algorithm

Algorithm 3 iterates all shared global values inside $\mathcal{I}_i$ and $\mathcal{I}_j$, for each pair of values v_i and v_j, it retrieves their types τ_i and τ_j (Line 2 - Line 4). Following the subtyping relations defined in Figure 6, if τ_i is a subtype if τ_j, i.e., $\tau_i \leq \tau_j$, we substitute v_j's type τ_j with τ_i and modify all v_j's usage accordingly. The situation when $\tau_j \leq \tau_i$ is handled similarly (Line 5 - Line 8).

Example 6.2. For the assembly code in Figure 2b, the two functions are placed into two IR partitions: $\mathcal{I}_{bar}$ and $\mathcal{I}_{foo}$. The global value `bar` is shared between $\mathcal{I}_{bar}$ and $\mathcal{I}_{foo}$. $\mathcal{I}_{bar}$ contains the definition of `bar` while $\mathcal{I}_{foo}$ contains a call site of `bar`. As explained in Example 6.1, the type of `bar` inside $\mathcal{I}_{bar}$ can be inferred as $\tau_1 = (\texttt{int}_{64}*, \texttt{int}_{32}, \texttt{int}_{32}, \texttt{int}_{32}, \texttt{int}_{32}, \texttt{int}_{32}, \texttt{int}_{32}) \rightarrow \texttt{void}$. However, since $\mathcal{I}_{foo}$ does not have any hints about `bar`'s first parameter `rdi`, `bar`'s type could be inferred as $\tau_2 = (\texttt{int}_{64}, \texttt{int}_{32}, \texttt{int}_{32}, \texttt{int}_{32}, \texttt{int}_{32}, \texttt{int}_{32}, \texttt{int}_{32}) \rightarrow \texttt{void}$. When linking $\mathcal{I}_{bar}$ and $\mathcal{I}_{foo}$, the algorithm could establish a subtype relation between τ_1 and τ_2 as $\tau_1 \leq \tau_2$. Therefore, τ_2 inside $\mathcal{I}_{foo}$ will be replaced by τ_1.

Line 9 - Line 11 in the algorithm handles the situation when the two types of the same global value do not form any subtyping relations. The algorithm merges the two types together into another type τ_m that satisfies $\tau_m \leq \tau_i$ and $\tau_m \leq \tau_j$. The type merging is similar to existing binary-level type inference techniques that create union or structure types [59, 116, 126]. The merging is guaranteed to succeed due to the following theorem.

Theorem 6.1. Given a a shared code entity v in IR partitions $\{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_n\}$, denoted as $\{v_1, v_2, ..., v_n\}$, whose types are $\{\tau_1, \tau_2, ..., \tau_n\}$, respectively. There always exists a type τ that satisfy $\tau \leq \tau_1 \land \tau \leq \tau_2 \land ... \land \tau \leq \tau_n$.

Proof. Consider the code entity in v in the monolithic IR $\mathcal{I}$, and suppose it has type τ, which implies that τ aggregates all v's usage $U = \{u_1, u_2, ..., u_n\}$ inside $\mathcal{I}$. Since any IR partition $\mathcal{I}_i$ contains a subset of instructions of $\mathcal{I}$'s, v's usage U_i inside $\mathcal{I}_i$ is always a subset of U, i.e., $U_i \subseteq U$. Therefore, τ can also aggregate U_i, and can safely substitute the type of v, τ_i, inside $\mathcal{I}_i$. According to the definition of subtyping [87], $\tau \leq \tau_i$. □

Theorem 6.1 implies that if the two types of the same global value do not form subtyping relations, we can always find another type τ that is a subtype of both of them. For example, consider two structures, $\tau_1 = x_1\left\{\mathtt{int}_{32}, \mathtt{int}_{64}, \mathtt{int}_{64}*\right\}$, and $\tau_2 = x_2\left\{\mathtt{int}_{32}, \mathtt{int}_{64}*, \mathtt{int}_{64}\right\}$. τ_1 and τ_2 do not form subtyping relations according to Figure 6. But we can find another type $\tau_3 = x_3\left\{\mathtt{int}_{32}, \mathtt{int}_{64}*, \mathtt{int}_{64}*\right\}$ that satisfies $\tau_3 \leq \tau_1 \wedge \tau_3 \leq \tau_2$.

Additionally, we argue that more sophisticated type inference techniques, such as the learning-based ones that require the presence of the whole program code [61, 76, 85], are often performed on the output of a binary lifter instead of the intermediate results during the lifting process. Therefore, DIATOM's polylithic design will not threaten its practicality.

7 Implementation

We implemented DIATOM in C/C++ on top of LLVM 3.6 [58]. We follow numerous open-source binary lifters such as RETDEC [57, 66], MCTOLL [92], and MCSEMA [31, 37] to implement basic lifting components such as the procedures at Line 2 and Line 8 in Algorithm 1.

Disassembly. The disassembly step is responsible for lifting the machine code into LLVM IR statements and constructing the inter-procedural control-flow graph (ICFG) from the input binary. DIATOM adopts the `Capstone` [2] library for instruction decoding and reuses the `Capstone2LlvmIR` [3] library from RETDEC [57] to translate assembly code into LLVM IR statements. DIATOM's general disassembly technique is based on the combination of recursive descent and linear sweep algorithms [80], and indirect branches or function calls are handled with data-flow analysis, similar to existing binary lifters [13, 34, 37, 38, 57, 92]. For example, for indirect jumps, DIATOM identifies the location of the jump table by performing backward data-flow analysis from the jump instruction[75, 114], which contains all jump target addresses. For calling to function pointers, DIATOM also performs backward data-flow analysis for identifying possible target callee functions in a best-effort way. Note that missing targets of function pointers will not cause missing control-flow edges on the ICFG, and such indirect calls are conservatively represented as abstract call sites, ensuring that ICFG remains sound without enumerating specific callees. The output of disassembly is an initial LLVM IR, and all the remaining lifting steps are performed on the IR instead of the assembly code. Note that the data-flow analysis used in disassembly is light-weight and has limited impact on the overall scalability, as shown in § 8.2.

IR Splitting. The IR splitting step clones IR functions from the monolithic LLVM IR module $\mathcal{I}_{dis}$ into individual IR modules $\{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_n\}$. Each function inside $\mathcal{I}_{dis}$ is traversed in parallel, for each IR statement, DIATOM first extracts the information it provides, such as the types of opcode and all operands. Then, it uses the `IR Builder` provided by LLVM to create the same statement in the target IR partition $\mathcal{I}_i$. DIATOM handles dependencies between IR values by maintaining a mapping that records correspondence between the values of two IR modules. If we encounter an operand that has not been copied, a placeholder value in the $\mathcal{I}_i$ is generated. Once the translation for that operand is completed, we replace all usages of the placeholder with the actual values in $\mathcal{I}_i$ [124].

Data-flow Recovery. The procedure at Line 8 in Algorithm 1 performs data-flow recovery on all IR partitions in parallel. Specifically, DIATOM recovers high-level constructs such as variables and function prototypes on the IRs. DIATOM adopts similar underlying algorithms for variable and function prototype recovery as existing binary lifters [13, 34, 37, 38, 57, 92]. Taking function prototype recovery as an example, the binary code passes parameters and returns values according to the architecture-specific calling conventions [69]. For example, the `X86_64`'s calling convention passes the first six integer arguments in general registers(e.g., `rdi`), and additional integer arguments are passed through the stack in reverse order. Traditional monolithic algorithms detect parameters of a function by performing a live variable analysis of register usage, along with a stack frame analysis [92], and taking registers that have no reaching definitions and stack frame accesses out of the current functions as the parameters. Those analysis results are used to analyze all the call sites to identify the passed values. In DIATOM, instead of reasoning about inter-procedural data-flows between functions directly, Diatom modifies the original algorithms by leveraging pre-generated summaries (§ 4) to identify values passed to callee functions at call sites.

Calls through function pointers pose additional challenge for parameter recovery, as the callee function is not statically known. In such cases, we follow existing binary lifters [34, 37, 57] to recover parameters by analyzing call site information directly. Specifically, we inspect the calling convention at each indirect call site and identify the registers and stack locations that are live and consumed at the call instruction. Additionally, it is also possible to further adopt other techniques [55, 56] that associate function pointers with callee functions, which enables the usage of callee-specific summaries or prototypes.

IR Optimization. Following existing lifters [31, 37, 57, 92, 130], Diatom applies LLVM optimizations such as `SimplifyCFG` to eliminate unnecessary code produced during the lifting process (Line 8 in Algorithm 1). Note that, from monolithic lifters that perform optimizations on the whole LLVM module sequentially, Diatom applies optimizations to individual IR partitions in parallel.

8 Evaluation

Our evaluation aims to answer the following research questions.

- **RQ1**: Can Diatom scale to extreme large binaries?
- **RQ2**: Can Diatom preserve the lifting soundness compared with monolithic lifters?
- **RQ3**: What are the contributions of Diatom's main components (ablation study)?

8.1 Experimental Setup

Environment Setup. We evaluate Diatom on an Intel Xeon(R) computer with a Platinum 8358 CPU and 512GB of memory running Ubuntu 18.04 LTS. The computer is equipped with 128 CPU cores, therefore, unless otherwise specified, Diatom is run with 128 threads.

Benchmarks. As shown in Table 1, we use 16 large `X86_64` binaries in the evaluation, including 13 binaries compiled from famous real-world open-source C/C++ projects such as `Firefox`, one pre-built library from TensorFlow [9], and two large binaries from a standard benchmark `x86-sok` [80]. For each binary, we adopt both its stripped and debug versions in the evaluation. Stripped binaries represent the most general case where the binary itself does not contain any high-level information, such as types and function prototypes. Moreover, since the benchmark programs are collected from different sources, different compilers and flags are used for generating the binaries. For example, `blender` is compiled with `clang-12`, `libtensorflow.so` is compiled with `gcc-7.3.1`, and `libv8.so` is compiled with `clang-6`. Various optimization levels are also used such as `-O3`, `-Ofast`, and `-Os`, etc.

We have several criteria when selecting the benchmarks: popular in the community, the sizes of codebases/compiled binaries are large enough, and widely used for evaluating previous scalable static code analysis methods. For example, `Linux` and `Firefox` are used for evaluating distributed inter-procedural data-flow analysis [104] and disk-based flow-/context-sensitive analysis [131]. `TensorFlow` is used to evaluate a new algorithm for parallelizing the CFG construction (disassembly) on binaries [72]. `FFmpeg`, `MySQL`, `Firefox`, and `V8` are used for evaluating scalable sparse value flow analysis [94, 95]. Several selected projects are even larger than all benchmarks used by previous work, such as `Blender` and `Oceanbase`.

Baseline Approaches. We compared Diatom with two state-of-the-art monolithic binary lifters as follows. We are also aware of other popular lifters such as mctoll [92, 119] and Reopt [4], however, they crash on all the benchmarks, therefore, we omit them in the evaluation.

- RetDec [57] is an open-source monolithic binary lifter with industrial strength and is widely adopted in numerous binary analysis tasks [41, 65, 66, 68, 77, 115].

Table 1. Lifting time (seconds) and the speedup over existing lifters on binaries with and without debug information. “-” means the lifter crashes on the binaries. The column **Code Size** is the size of the code section in the compiled binaries, the numbers are shown in megabytes (MB).

Project	Binary	Code Size	w/ debug info (seconds)					w/o debug info (seconds)				
			DIATOM	PLANKTON		RETDEC		DIATOM	PLANKTON		RETDEC	
			Time	Time	Speedup	Time	Speedup	Time	Time	Speedup	Time	Speedup
RocksDB	rocksdb.so	6	28.61	177.03	6.2x	385.38	13.5x	25.26	166.13	6.6x	382.98	15.2x
PHP	php	9	41.04	224.64	5.5x	832.61	20.3x	41.21	219.91	5.3x	799.94	19.4x
MySQL	mysqld	11	51.53	337.93	6.6x	1114.7	21.6x	47.89	406.4	8.5x	1170.23	24.4x
FFmpeg	ffmpeg	15	81.46	621.81	7.6x	655.87	8.1x	77.1	565.03	7.3x	676.89	8.8x
MariaDB	mariadbd	19	88.87	459.77	5.2x	3212.08	36.1x	73.9	489.07	6.6x	3250.43	44.0x
V8	libv8.so	24	101.96	797.43	7.8x	12519.84	122.8x	92.25	656.28	7.1x	10389.06	112.6x
GCC	cc1	27	457.73	2981.81	6.5x	3719.29	8.1x	316.85	3046.21	9.6x	3914.21	12.4x
GCC	cc1plus	30	489.08	3145.66	6.4x	4367.81	8.9x	338.42	3238.07	9.6x	4672.6	13.8x
NodeJS	node	36	191.78	1793.86	9.4x	-	-	171.37	2044.23	11.9x	-	-
Linux	vmlinux	127	590.41	3707.85	6.3x	-	-	574.32	3842.87	6.7x	-	-
FireFox	libxul.so	127	668.7	4853.96	7.3x	-	-	581.86	4254.06	7.3x	-	-
Chrome	chrome	164	997.96	5562.91	5.6x	-	-	914.51	5732.76	6.3x	-	-
TensorFlow	tensorflow.so	185	1079.81	8399.67	7.8x	-	-	1134.29	8709.55	7.7x	-	-
LLVM	clang-20	189	2133.79	10534.34	4.9x	-	-	880.06	8090.96	9.2x	-	-
Blender	blender	209	2424.24	16452.05	6.8x	-	-	1557.74	26130.54	16.8x	-	-
OceanBase	observer	551	2562.46	17802.23	6.9x	-	-	2509.58	13735.62	5.5x	-	-
Average	-	108	**749.34**	4865.81	**6.7x**	3350.95	**29.9x**	**583.54**	5082.98	**8.2x**	3157.04	**31.3x**

- PLANKTON [130] is a most recent monolithic binary lifter[1]. Similar to RETDEC, PLANKTON is a general binary lifter that handles both stripped and unstripped binaries. It also features several new algorithms to produce analysis-friendly LLVM IRs by leveraging debug information.

8.2 RQ1: Scalability

In this section, we evaluate the scalability of DIATOM in terms of both the parallel speedup and the memory consumption.

Speedup. We first compare DIATOM with two state-of-the-art monolithic binary lifters to show its parallel speedup. Table 1 shows the time consumption of DIATOM, PLANKTON [130], and RETDEC [57] on 16 benchmarks, ordered by the code size. The table shows the results of lifting binaries with and without debug information to demonstrate the generality of DIATOM. RETDEC is only able to lift the smallest 8 benchmarks and crashes on the others (no valid LLVM IRs produced), therefore, some results are omitted in Table 1.

The results show that DIATOM achieves an average speedup of 7.45× over the monolithic lifter PLANKTON on all binaries (with and without debug information), and the speedup can be as high as 16.8×. Among the 8 benchmarks that can be handled by RETDEC, DIATOM achieves an average speedup of 30.6×. It is noteworthy that even a 7.45× speedup is significant enough to make binary lifting useful in practice. For example, originally, it takes nearly 5 hours to process the `observer` binary. With the new design, it saves more than 4 hours, making the binary lifting acceptable in the industrial setting. The speedup over RETDEC is much higher than that over PLANKTON mainly because RETDEC runs much slower than PLANKTON. For example, it takes PLANKTON only 797.43 seconds to lift the debug version of `libv8.so` binary, while RETDEC takes 12519.84 seconds.

For small-sized binaries such as `php` and `mysqld` (dozens of megabytes), PLANKTON could finish lifting within 10 minutes, which is already very fast. But DIATOM can still achieve 5× to 6× speedup by further shortening the lifting time to less than 1 minute. The scalability problem in monolithic lifters becomes much more severe on bigger-sized binaries (hundreds of megabytes to a few gigabytes) such as `clang-20`. PLANKTON and RETDEC become orders-of-magnitude more expensive

[1] Available at https://www.clearblueinnovations.org/

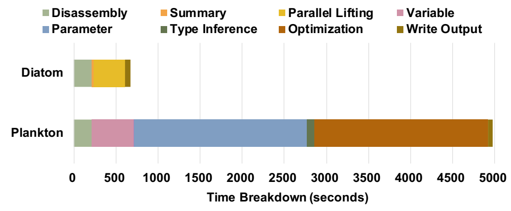


Fig. 7. Average time breakdown of different lifting stages on all binaries.

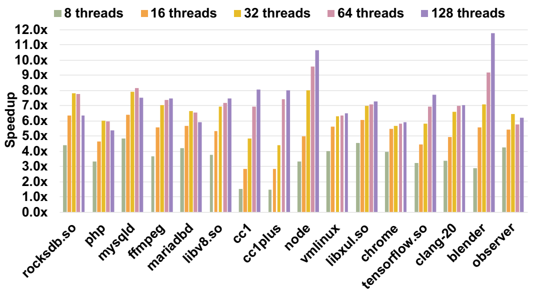


Fig. 8. Average speedup growth with threads.

and even require a few hours to finish the lifting process. Thanks to the new polylithic design, Diatom is much more scalable than the compared monolithic lifters when handling extremely large binaries. For example, Plankton could take over 7 hours to finish translating `blender` without debug information, while Diatom only takes about 25 minutes, achieving a 16.5× speedup. As shown in the table, Diatom finishes lifting all binaries within 1 hour, which is significant enough to support upper layer analysis on programs with millions of lines of code.

The overall results also follow the principle that larger binaries take a longer lifting time, which is accordant with our observation in § 5. However, there are a few violations in Table 1, for example, Plankton spends more time on `ffmpeg` (15MB code size) than `mariadbd` (19MB code size). The lifting process involves numerous analyses, such as data-flow analysis, whose time complexity is not solely determined by the code size. Therefore, it is possible that some smaller-sized binaries take longer lifting time. Such a phenomenon can also be observed in numerous previous works [94, 125, 130], i.e., analyses on smaller codebases could take more time than those on larger codebases. Since Diatom only tries to parallelize the original lifting process without changing the internal algorithms, a similar phenomenon could also happen in its results.

Runtime breakdown. Figure 7 further presents a comparison between Diatom and Plankton in terms of the average time breakdown of different lifting stages on all binaries. The figure shows that Diatom has a much shorter total lifting time than Plankton (666.4s vs. 4974.4s on average). While the disassembly step only accounts for about 4% of the total lifting time in Plankton, it takes about 32% of the time in Diatom. It is because Diatom still performs the disassembly step in a monolithic way, as the overall performance is improved, the monolithic step now becomes a performance bottleneck, which can be further improved with existing parallelism techniques [72]. The summary generation is also very efficient for practical usage, taking only 3.2% of the total lifting time. The parallel lifting step in Diatom takes most of the time (54%) because it includes multiple stages, such as IR splitting, IR linking, and code optimizations.

Threads vs. speedups. We also studied the relationship between the number of threads and the speedups. As shown in Figure 8, the overall speedup increases with the growth of the number of threads. On average, Diatom achieves a speedup of 3.56×, 5.14×, 6.54×, 7.19× and 7.45× with 8, 16, 32, 64, and 128 threads.

More formally, the observed speedup can be calculated as follows.

$$S(N) = \frac{T_{serial} + T_{parallel}}{T_{serial} + \frac{T_{parallel}}{N} + T_{overhead}} \tag{1}$$

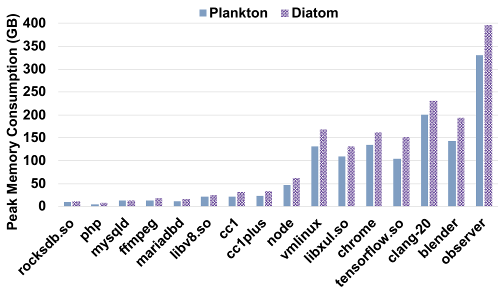


Fig. 9. Peak memory consumption (GB).

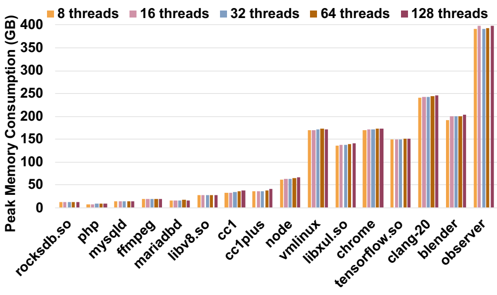


Fig. 10. Peak memory with number of threads.

where $S(N)$ is the speedup under N threads, T_{serial} is the time spent on lifting stages that have to be done in serial (e.g., disassembly), $T_{parallel}$ is the time spent on lifting stages that are parallelized (e.g., information recovery) by DIATOM, and $T_{overhead}$ is the time spent on extra overhead for parallelism (e.g., IR splitting). Therefore, as N grows, $\frac{T_{parallel}}{N}$ shrinks, but $T_{overhead}$ may increase or stay nontrivial, limiting the parallelism gain or even hurting performance. When overhead grows faster than the benefit from parallel division, adding threads yields little to no speedup. In extreme cases, it causes a slowdown.

For example, for small-sized binaries, the performance could degrade when we use too many threads since the amount of computation to be done is so small that the performance benefit of parallelizing lifting stages is no longer superior to the increased splitting/merging cost among threads. It is also possible that the speedup growth plateaus at one point with respect to the number of threads. For example, for `rocksdb.so`, the highest speedup is achieved with 32 and 64 threads (7.8×), while the speedup degrades to only 6.4× using 128 threads. On the contrary, for bigger-sized binaries, using more threads is more likely to achieve higher speedup, for example, DIATOM achieves the highest speedup of 11.8× with 128 threads on `blender`, while the speedup is only 9.2× when using 64 threads. This phenomenon has also been observed in previous efforts on parallelizing static analysis [71, 104]. This implies that in practice, we can seek a sweet spot of parallelism for different subjects according to the runtime trend as the number of workers changes.

Memory consumption. Figure 9 shows a comparison of the peak memory consumption (GB) between DIATOM and PLANKTON on all benchmarks (average over binaries with and without debug information). On average, DIATOM consumes 22.6% more memory than PLANKTON. The main reason for extra memory consumption is that DIATOM splits and clones the original monolithic LLVM module into multiple partitions that do not have overlapping function definitions (but might have overlapping function declarations), which requires more memory. There are two aspects of extra memory consumption: (1) different IR partitions have independent environments (containing type mappings and def-use chains on constants), which brings some memory overhead. (2) Although partitions do not have overlapping function definitions, they might have shared function declarations or global variables to keep the validity of the IR, which also brings some memory overhead. Figure 10 further shows the peak memory consumption given different numbers of threads. As illustrated in the figure, the peak memory consumption of DIATOM is fairly constant across different numbers of threads.

We would like to argue that slightly higher memory consumption of DIATOM does not hurt its practicality due to three reasons. First, higher memory requirements are easier to meet with vertical scaling (aka scaling up) [91] by adding additional hardware resources to the system. By contrast, higher speedup cannot be simply achieved by adding more computing resources without

Table 2. Soundness-preserving ratios. Diatom-NS denotes a variant of Diatom that does not leverage data-flow summaries during parallel lifting. Diatom-NT does not perform type-aware IR linking. The results are presented as r_1/r_2, which denotes the portion of global values inside polylithic lifting results that are exactly the same as the monolithic lifting (r_1) and those that are over-approximated results (r_2).

Binary	w/ debug info						w/o debug info					
	Diatom		Diatom-NS		Diatom-NT		Diatom		Diatom-NS		Diatom-NT	
	Data-flow	Type	Data-flow	Type	Data-flow	Type	Data-flow	Type	Data-flow	Type	Data-flow	Type
php	1.0/1.0	1.0/1.0	.98/1.0	1.0/1.0	1.0/1.0	.98/.98	.98/1.0	1.0/1.0	.86/.93	.94/1.0	.98/1.0	.78/.78
rocksdb.so	.98/1.0	.99/1.0	1.0/1.0	1.0/1.0	.98/1.0	.99/.99	.99/1.0	.99/1.0	.81/.88	.97/.98	.99/1.0	.76/.76
ffmpeg	1.0/1.0	1.0/1.0	1.0/1.0	.99/.99	1.0/1.0	.97/.97	.94/1.0	1.0/1.0	.77/.84	.96/.97	.94/1.0	.77/.77
mysqld	.98/1.0	1.0/1.0	.95/1.0	1.0/1.0	.98/1.0	.99/.99	.92/1.0	.99/1.0	.62/.69	.96/.96	.92/1.0	.86/.86
cc1	1.0/1.0	1.0/1.0	1.0/1.0	.97/.99	1.0/1.0	.99/.99	1.0/1.0	.99/1.0	.35/.38	.94/.94	1.0/1.0	.82/.82
cc1plus	1.0/1.0	1.0/1.0	1.0/1.0	.98/.99	1.0/1.0	.99/.99	1.0/1.0	1.0/1.0	.34/.38	.94/.95	1.0/1.0	.82/.83
mariadbd	1.0/1.0	1.0/1.0	.99/1.0	1.0/1.0	1.0/1.0	1.0/1.0	.94/1.0	1.0/1.0	.88/.94	.97/.99	.94/1.0	.85/.85
libv8.so	1.0/1.0	.99/1.0	1.0/1.0	.95/.97	1.0/1.0	.99/1.0	1.0/1.0	1.0/1.0	.55/1.0	.95/.98	1.0/1.0	.83/.83
node	1.0/1.0	1.0/1.0	.99/.99	1.0/1.0	1.0/1.0	1.0/1.0	.96/1.0	1.0/1.0	.70/.73	.99/.99	.96/1.0	.81/.81
tensorflow	.96/1.0	.97/.97	.69/.77	.91/.92	.96/1.0	.83/.84	.96/1.0	.97/.97	.68/.77	.91/.92	.96/1.0	.73/.74
vmlinux	1.0/1.0	1.0/1.0	1.0/1.0	.99/.99	1.0/1.0	.99/.99	1.0/1.0	1.0/1.0	.28/.30	.99/.99	1.0/1.0	.81/.81
libxul.so	1.0/1.0	1.0/1.0	.99/1.0	.96/.97	1.0/1.0	.98/.99	.91/1.0	1.0/1.0	.52/.57	.95/.97	.91/1.0	.65/.65
blender	.98/1.0	.99/.99	1.0/1.0	1.0/1.0	.98/1.0	.83/.84	.89/1.0	.99/1.0	.85/.93	1.0/1.0	.89/1.0	.83/.84
clang-20	.99/1.0	1.0/1.0	.95/.97	.96/.97	.99/1.0	1.0/1.0	.99/1.0	1.0/1.0	.85/.90	.95/.97	.99/1.0	.93/.93
observer	.92/1.0	1.0/1.0	.71/.96	.98/.99	.92/1.0	.95/.95	.97/1.0	1.0/1.0	.64/.88	.98/.99	.97/1.0	.87/.87
chrome	1.0/1.0	.99/1.0	1.0/1.0	.95/1.0	1.0/1.0	.59/.59	.99/1.0	1.0/1.0	.98/.99	.95/.99	.99/1.0	.57/.57
Average	**.99/1.0**	**1.0/1.0**	**.95/.98**	**.98/.99**	**.99/1.0**	**.94/.94**	**.97/1.0**	**1.0/1.0**	**.67/.76**	**.96/.97**	**.97/1.0**	**.79/.80**

an effective parallelism schema. Second, as shown in Figure 10, the peak memory consumption of Diatom is fairly constant across different numbers of threads, therefore, it is possible to achieve more speedup without incurring higher memory overhead. Third, since the proposed polylithic design performs binary lifting on IR partitions in an asynchronous way, it has a great potential to be further improved for reducing memory consumption using disk-based systems [131] or distributed environments [104]. We leave it as one of our future works.

8.3 RQ2: Soundness

In this section, we evaluate whether Diatom is soundness-preserving. We use the monolithic design-based binary lifter Plankton as the translation ground truth and evaluate whether Diatom can preserve the lifting soundness in terms of data-flows and types. The control-flow soundness is naturally guaranteed since Diatom still performs control-flow recovery in a monolithic way, therefore, it produces the same results as Plankton. Therefore, we only evaluate whether Diatom can preserve data-flow and type soundness in the lifted IR compared with Plankton.

The evaluation is performed through *IR comparison*, i.e., we directly compare the IR code lifted from binaries. More specifically, for an input binary B, suppose the monolithic lifter Plankton produces the IR $\mathcal{I}_{mono}$, and Diatom produces $\mathcal{I}_{poly}$. To evaluate the data-flow soundness, for each function f_p inside $\mathcal{I}_{poly}$, we find its corresponding function f_m inside $\mathcal{I}_{mono}$ by examining whether they have the same assembly address. As discussed in § 4, low-level machine code leverages registers and stack memory to implement inter-procedural data-flows, and the lifter should translate them into high-level formal parameters and return values. Therefore, we check whether f_p and f_m have the same set of formal parameters and return values to validate data-flow soundness. For type soundness, we check whether the types of global variables and function prototypes inside $\mathcal{I}_{poly}$ can be safely used by their counterparts inside $\mathcal{I}_{mono}$. Table 2 shows the results of the soundness-preserving ratios for lifting binaries with and without debug information in the form of r_1/r_2 (different values whose maximum values are both 1.0), which denotes the portion of global values (functions and global variables) inside polylithic lifting results that are the same as the monolithic lifting (r_1) and those that are over-approximated results (r_2). The over-approximation means that Diatom could recover more information than monolithic lifting but does not miss any. For example, f_p contains over-approximated data-flows over f_m if f_p has a longer formal parameter list. τ_p is an over-approximation of τ_m if $\tau_p \preceq \tau_m$.

As shown in Table 2, in both scenarios (with and without debug information), Diatom is soundness-preserving in terms of data-flows and types. On average, over 98% of functions lifted by Diatom exhibit exactly the same data-flows as Plankton, and the percentage reaches 100% when including functions with over-approximated data-flows. The table also shows that Diatom is able to preserve the type soundness for all binaries, achieving a 100% exact match rate on average. The soundness-preserving ratios for lifting binaries with debug information are also slightly higher than that of binaries without debug information (e.g., 99% vs. 97% on exact matched data-flows). It is because the debug information provides rich hints about function prototypes and types, which lets the lifter recover accurate information without additional analysis, thus reducing the difference between whole program analysis and separated analysis. Note that even when binary is built with debug information enabled, some functions still lack prototype information because of inlined library functions [11], incomplete debug information [35], or function outlining [129].

We found that the majority of over-approximated data-flows are external functions that do not have function bodies. For those functions, Diatom is unable to generate data-flow summaries, and the prototypes of those external functions can only be indirectly inferred by analyzing all their call sites. Therefore, when analyzing call sites for the same function in different partitions, Diatom could over-approximate some of them, resulting in over-approximated data-flows in the linked IR. In terms of type soundness, we found that there are two main kinds of over-approximated global types, one occurs between pointer and integer types, and the other occurs between different pointer types. Both of them are caused by inaccurate function parameter recovery for external functions. For example, the global variable `kUnknownFileChecksumFuncName` inside the `rocksdb.so` binary has only one use site, which is used as the parameter for calling functions of C++ string libraries. However, without data-flow summaries of function bodies, calls to external functions are analyzed separately in different partitions, causing possible inaccurate call site parameters on used global variables. Therefore, the inferred types for such global variables might be over-approximated (e.g., inferred integer-typed global variable as a pointer type). Across the `rocksdb.so` binary, there are more than 13,000 call sites invoking C++ string library functions, which amplifies the impact of conservative prototype inference for these external calls. Despite these cases, the overall impact remains limited. As shown in Table 2, the proportion of over-approximated global types is very small (less than 1%); therefore, it will not hurt the quality of results.

8.4 RQ3: Ablation Study

In this section, we conduct ablation studies that involve three variants of Diatom to justify our design choices and quantify the influence of our key algorithms.

Lock-free IR cloning. We first study the runtime overhead of Diatom's lock-free IR cloning (§ 5). Figure 11 shows a comparison between the time spent on IR splitting for all 16 benchmarks under the same function distribution results with 64 threads, where the `naive` label represents the IR splitting method in the LLVM framework [8, 50, 112] as described in § 5. The figure shows that Diatom achieves an average speedup of 11.1× on all benchmarks compared with the naive IR splitting method. On average, Diatom takes only 86 seconds to split the initial IR into 64 partitions, while the naive method takes over 1085 seconds, which could severely affect the overall performance of polylithic lifting. Since Diatom creates an independent environment for each newly created IR partition, the splitting procedure can be fully parallelized on each partition, which significantly reduces the runtime overhead.

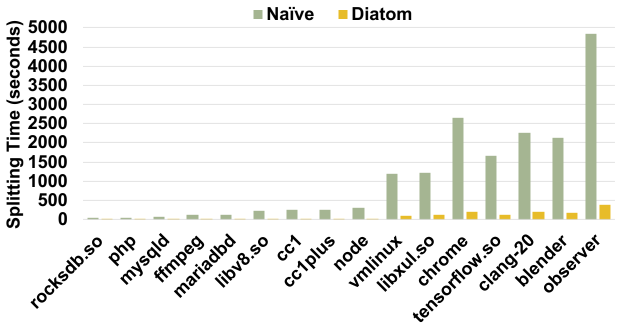


Fig. 11. Comparison of the time spent on IR splitting by Diatom and LLVM's naive splitting method.

Data-flow summary. To demonstrate the effectiveness of data-flow summaries, we propose an ablation of Diatom that does not generate data-flow summaries during polylithic lifting, denoted as Diatom-NS. As shown in Table 2, when handling binaries with debug information, Diatom-NS loses the data-flow soundness for 4% more functions compared with Diatom. When handling binaries without debug information, while Diatom still preserves 100% data-flow soundness, Diatom-NS only maintains 76% of them. For example, Diatom-NS performs badly on `vmlinux` without debug information (achieving only 28% data-flow soundness). `vmlinux` contains a lot of "wrapper functions", which typically refer to a function that encapsulates or provides an interface to another underlying function or system call. Without data-flow summaries, those wrapper functions cannot be properly analyzed for recovering parameters. It is because there might not be explicit load instructions for parameter registers, which require information from other functions. Therefore, when functions are split into partitions, they cannot be correctly analyzed without data-flow summaries. Binary lifting has a very low tolerance for losing soundness because the downstream analysis can always refine the lifting results with more precise analyses but cannot compensate for missed data-flows [56, 108, 110]. Therefore, the data-flow summary is crucial for preserving the data-flow soundness in Diatom. Additionally, losing data-flow soundness can decrease the type soundness since type inference requires reasoning about data-flows for global values. As shown in the table, Diatom-NS's type soundness is 3% lower than Diatom on average.

Type-aware IR linking. We further studied how the type-aware IR linking helps preserve the type soundness. We propose another ablation of Diatom that does not perform type-aware IR linking, denoted as Diatom-NT. As shown in Table 2, although Diatom-NT has the same level of data-flow soundness as Diatom, it has a much lower type soundness ratio. On average, Diatom-NT only preserves the type soundness for about 87% of global values. Debug information provides explicit types for quite a few global values; therefore, Diatom-NT performs better on binaries with debug information compared with those without. The results demonstrate the importance of type-aware IR linking. Without type soundness, downstream static analysis that relies on type information will not work correctly [130].

9 Discussion

Generality. Although experiments are performed on `X86_64` binaries, Diatom does not assume any specific instruction set architectures (ISAs) or calling conventions. There are two main reasons. First, one of the most important advantages of binary lifters is that they are able to use a unified intermediate representation (IR) to represent different ISAs. The disassembly step already translates the binary code into ISA-agnostic LLVM IRs so that later steps do not need to handle complex instruction details. According to DIATOM's design, the disassembly step is performed in the original monolithic way, and it only focuses on accelerating later stages that are performed on the IR code. Therefore, other ISAs can be accommodated in the same way as X86_64. Second, DIATOM's algorithm is established on the abstract language defined in § 3, which is also ISA and calling convention agnostic. Therefore, DIATOM can be applied to other scenarios without performance loss, and the parallelism can also bring about speedup compared with single-thread lifting.

Limitations. DIATOM still has several limitations. First, it works on a fixed control-flow graph recovered by existing algorithms. As discussed in § 1, it is generally undecidable to achieve perfect lifting soundness (e.g., for function boundaries recovery), therefore, it is possible that some function boundaries are misidentified, but the generated summaries guarantee that the final lifting results are the same as the monolithic lifting. Second, our current implementation follows common algorithms of existing lifters, which do not incorporate all advanced binary analysis techniques, such as function pointer analysis [55, 56], which might affect the adoption of proposed techniques. For example, the type-aware linking algorithm does not handle aliasing function pointers. It currently only concerns about global values that share the same symbol name (§ 6). However, we believe that DIATOM has great potential to further benefit other sophisticated analyses. On the one hand, DIATOM can work together with advanced control-flow recovery (disassembly) techniques [17, 42, 54–56, 86, 89, 122, 123] that require whole program analysis since the polylithic design still performs the disassembly step in the traditional monolithic way. On the other hand, principled binary-level information recovery methods such as probabilistic type analysis [59, 116, 126] and function pointer analysis [55, 56] usually require the presence of already lifted IRs, therefore, DIATOM could also act as the pre-analysis for those techniques. Third, the current implementation of DIATOM still does not have enough pre-knowledge about external functions, which causes some inaccuracy (as shown in § 8.3). This problem could be easily fixed by analyzing debug information of external libraries during binary lifting, and we leave it as future work.

10 Related Work

Binary Lifting. Numerous binary lifters have been proposed to translate binaries into high-level IRs such as LLVM. SECONDWRITE [15, 16, 38] uses VSA-based approaches to recover variables and data types from stripped binaries. POLYNIMA [32], BINREC [13, 84] lift binaries based on dynamic disassembly. LASAGNE [92] statically translates X86_64 binaries to LLVM IR and then compiles it to Arm while enforcing the x86 memory ordering model. REVNG [33, 34] relies on QEMU to perform lifting. LISC [47] automatically learns translation rules from assembly to IR. MCTOLL [5] and RETDEC [6] also adopt code optimizations to refine the produced LLVM IR. PLANKTON [130] produces high-quality IRs from binaries with debug information to enable precise static analysis.

Parallel Binary Translation. Despite efforts on binary lifting, very few studies have been conducted on the scalability issue. One of the related efforts is proposed by Meng et al. [72]; they designed a new algorithm for parallelizing the CFG construction (disassembly). However, as discussed in § 1, the disassembly step only accounts for a small portion of the lifting time. Therefore, it cannot be used to solve the scalability problem in binary lifting. Another closely related work is B2R2 [51], which also proposes a parallel binary lifting technique that accumulates a number of decoded instructions and asynchronously lifts and optimizes them on multiple threads. However, there are two main differences between our work and B2R2. First, B2R2 aims to lift binaries into a custom IR named LowUIR, which cannot be directly applied to LLVM-based lifting due to the global shared context. Second, it does not perform the remaining steps of lifting other than disassembly. By contrast, we propose a general algorithm for parallelizing the entire lifting process.

Summary-based Program Analysis. Existing efforts use summary-based techniques to improve the precision or the scalability of program analysis. Theoretical foundations for summary-based analysis are explored in [23, 24, 46, 52, 88]. Dillig et al. [36] proposed a summary-based flow-sensitive analysis for verifying program memory safety properties. Sparse value-flow analysis [94, 103, 106, 120] and data-flow analysis [62, 78, 105, 118] use summary-based approaches to realize inter-procedural analysis and to avoid the re-analysis of the procedure body while enabling context sensitivity. Vardoulakis et al. [107] describe a summary-based CFA with a degree of flow sensitivity. Stein et al. [101, 102] describe a framework for interactive abstract interpretation that simultaneously supports the compositional application of procedure summaries. Android native code analysis [60, 81, 111, 113] uses summaries to reason about inter-language data-flow information. DDA [45] uses symbolic execution to compute a summary for each procedure that maps each parameter to the set of constant offsets that are used in dereferences of the parameter's value. The summary-based approach also enables modular/parallel program analysis. ThinLTO [50] computes simple summaries for each function, such as the number of instructions to perform cross-module optimization in parallel during compilation. Shi et al. [96] propose to partition the analysis task of each function into multiple sub-tasks to generate pipelineable function summaries. Bolt [12] proposes a MapReduce-style parallelism to scale the top-down summary-based analysis. As discussed in § 1.4, the problem of binary lifting is fundamentally different from static analysis.

11 Conclusion

This paper presents the design and implementation of a novel parallel binary lifter Diatom that follows a new polylithic design. We conduct experiments to show that Diatom can scale to large real-world binaries while still preserving the lifting soundness.

Data-Availability Statement

We will make Diatom available on GitHub at https://github.com/seviezhou/DiatomLifter.

Acknowledgments

We thank the anonymous reviewers for valuable feedback on earlier drafts of this paper, which helped improve its presentation. This work is funded by research grants from Ant Group and ByteDance.

References

[1] 2003. IDA Pro. https://www.hex-rays.com/ida-pro/.
[2] 2018. Capstone. https://www.capstone-engine.org/.
[3] 2019. Capstone2LlvmIR. https://github.com/chubbymaggie/capstone2llvmir/.
[4] 2020. Galois Inc. reopt. https://github.com/GaloisInc/reopt/.
[5] 2020. LLVM-mctoll. https://github.com/Microsoft/llvm-mctoll/.
[6] 2020. RetDec. https://github.com/avast/retdec/.
[7] 2021. Ghidra. https://github.com/NationalSecurityAgency/ghidra/.
[8] 2024. LLVM SplitModule. https://llvm.org/doxygen/SplitModule_8cpp.html.
[9] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.
[10] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.
[11] Toufique Ahmed, Premkumar Devanbu, and Anand Ashok Sawant. 2021. Finding Inlined Functions in Optimized Binaries. *arXiv preprint arXiv:2103.05221* (2021).
[12] Aws Albarghouthi, Rahul Kumar, Aditya V Nori, and Sriram K Rajamani. 2012. Parallelizing top-down interprocedural analyses. *ACM SIGPLAN Notices* 47, 6 (2012), 217–228.
[13] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, et al. 2020. BinRec: dynamic binary lifting and recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[14] Xiaoxin An, Freek Verbeek, and Binoy Ravindran. 2022. DSV: disassembly soundness validation without assuming a ground truth. In *NASA Formal Methods Symposium*. Springer, 636–655.
[15] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 295–308.
[16] Kapil Anand, Matthew Smithson, Aparna Kotha, Khaled Elwazeer, and Rajeev Barua. 2010. Decompilation to compiler high IR in a binary rewriter. *University of Maryland, Tech. Rep* (2010).
[17] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 177–189.
[18] Luitpold Babel, Hans Kellerer, and Vladimir Kotov. 1998. The k-partitioning problem. *Mathematical Methods of Operations Research* 47 (1998), 59–82.
[19] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*. Springer, 5–23.
[20] Gogul Balakrishnan and Thomas Reps. 2007. Divine: Discovering variables in executables. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 1–28.
[21] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*. 845–860.
[22] Joshua Bundt, Michael Davinroy, Ioannis Agadakos, Alina Oprea, and William Robertson. 2023. Black-box attacks against neural binary function detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 1–16.
[23] David Callahan. 1988. The program summary graph and flow-sensitive interprocedual data flow analysis. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. 47–56.
[24] David Callahan, Keith D Cooper, Ken Kennedy, and Linda Torczon. 1986. Interprocedural constant propagation. *ACM SIGPLAN Notices* 21, 7 (1986), 152–161.
[25] Satish Chandra and Thomas Reps. 1999. Physical type checking for C. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 66–75.
[26] Ligeng Chen, Zhongling He, and Bing Mao. 2020. CATI: Context-Assisted Type Inference from Stripped Binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 88–98.
[27] Maria Christakis, Thomas Cottenier, Antonio Filieri, Linghui Luo, Muhammad Numair Mansur, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. 2022. Input splitting for cloud-based static application security testing platforms. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1367–1378.
[28] Cristina Cifuentes and Antoine Fraboulet. 1997. Intraprocedural static slicing of binary executables. In *1997 Proceedings International Conference on Software Maintenance*. IEEE, 188–195.
[29] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception:{System-Wide} Security Testing of {Real-World} Embedded Systems Software. In *27th USENIX Security Symposium (USENIX Security 18)*. 309–326.
[30] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.
[31] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S Adve, and Christopher W Fletcher. 2020. Scalable validation of binary lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 655–671.
[32] Chinmay Deshpande, Fabian Parzefall, Felicitas Hetzelt, and Michael Franz. 2024. Polynima: Practical Hybrid Recompilation for Multithreaded Binaries. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1126–1141.
[33] Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. 2018. rev. ng: A multi-architecture framework for reverse engineering and vulnerability discovery. In *2018 International Carnahan Conference on Security Technology (ICCST)*. IEEE, 1–5.
[34] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev. ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*. 131–141.
[35] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. 2021. Who's debugging the debuggers? exposing debug information bugs in optimized binaries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1034–1045.
[36] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and compact modular procedure summaries for heap manipulating programs. *ACM SIGPLAN Notices* 46, 6 (2011), 567–577.

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

[38] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation.* 51–60.

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

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

[41] Javier Escalada, Ted Scully, and Francisco Ortin. 2021. Improving type information inferred by decompilers with supervised machine learning. *arXiv preprint arXiv:2101.08116* (2021).

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

[43] Behrad Garmany, Martin Stoffel, Robert Gawlik, and Thorsten Holz. 2019. Static detection of uninitialized stack variables in binary code. In *European Symposium on Research in Computer Security.* Springer, 68–87.

[44] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 1069–1084.

[45] Denis Gopan, Evan Driscoll, Ducson Nguyen, Dimitri Naydich, Alexey Loginov, and David Melski. 2015. Data-delineation in software binaries and its application to buffer-overrun discovery. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 145–155.

[46] Sumit Gulwani and Ashish Tiwari. 2007. Computing procedure summaries for interprocedural analysis. In *European Symposium on Programming.* Springer, 253–267.

[47] Niranjan Hasabnis and R Sekar. 2016. 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.* 311–324.

[48] Manuel Iori. 2005. Metaheuristic algorithms for combinatorial optimization problems. *4OR* 3 (2005), 163–166.

[49] Alan Jaffe, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. 2018. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension.* 20–30.

[50] Teresa Johnson, Mehdi Amini, and Xinliang David Li. 2017. ThinLTO: scalable and incremental LTO. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE, 111–121.

[51] Minkyu Jung, Soomin Kim, H Han, Jaeseung Choi, and Sang Kil Cha. 2019. B2R2: Building an efficient front-end for binary analysis. In *Proceedings of the NDSS Workshop on Binary Analysis Research.*

[52] John B Kam and Jeffrey D Ullman. 1976. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)* 23, 1 (1976), 158–171.

[53] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungll Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing intermediate representations for binary analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 353–364.

[54] Soomin Kim, Hyungseok Kim, and Sang Kil Cha. 2023. FunProbe: probing functions from binary code through probabilistic analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 1419–1430.

[55] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining indirect call targets at the binary level. In *Network and Distributed System Security Symposium, NDSS.*

[56] Sun Hyoung Kim, Dongrui Zeng, Cong Sun, and Gang Tan. 2022. BinPointer: towards precise, sound, and scalable binary-level pointer analysis. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction.* 169–180.

[57] Jakub Křoustek, Peter Matula, and Petr Zemek. 2017. Retdec: An open-source machine-code decompiler. *July 2018* (2017).

[58] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[59] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. (2011).

[60] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. 2020. Broadening horizons of multilingual static analysis: Semantic summary extraction from C code for JNI program analysis. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering.* 127–137.

[61] Daniel Lehmann and Michael Pradel. 2022. Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries. In *2022 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[62] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2024. SPATA: Effective OS Bug Detection with Summary-Based, Alias-Aware and Path-Sensitive Typestate Analysis. *ACM Transactions on Computer Systems* (2024).

[63] Yan Lin and Debin Gao. 2021. When function signature recovery meets compiler optimization. In *2021 IEEE Symposium on Security and Privacy (SP).* IEEE, 36–52.

[64] Cullen Linn, Saumya Debray, Gregory Andrews, and Benjamin Schwarz. 2004. Stack analysis of x86 executables. *Manuscript. April* (2004).

[65] Zhibo Liu and Shuai Wang. 2020. 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.* 475–487.

[66] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. 2022. SoK: Demystifying Binary Lifters Through the Lens of Downstream Applications. In *2022 2022 IEEE Symposium on Security and Privacy (SP)(SP). IEEE Computer Society, Los Alamitos, CA, USA.* 453–472.

[67] Kangjie Lu. 2023. Practical program modularization with type-based dependence analysis. In *2023 IEEE Symposium on Security and Privacy (SP).* IEEE, 1256–1270.

[68] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. 2022. The Convergence of Source Code and Binary Vulnerability Discovery–A Case Study. In *ASIACCS 2022, 17th ACM ASIA Conference on Computer and Communications Security, 30 May-3 June 2022, Nagasaki, Japan.*

[69] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2013. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft v0* 99, 2013 (2013), 57.

[70] Leandro TC Melo, Rodrigo G Ribeiro, Breno CF Guimarães, and Fernando Magno Quintão Pereira. 2020. Type inference for c: Applications to the static analysis of incomplete programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 3 (2020), 1–71.

[71] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications.* 428–443.

[72] Xiaozhu Meng, Jonathon M Anderson, John Mellor-Crummey, Mark W Krentel, Barton P Miller, and Srđan Milaković. 2021. Parallel binary code analysis. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* 76–89.

[73] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the 26th USENIX Security Symposium (SEC'17).* USENIX Association.

[74] Jiang Ming, Dongpeng Xu, and Dinghao Wu. 2015. Memoized semantics-based binary diffing with application to malware lineage inference. In *IFIP International Information Security and Privacy Conference.* Springer, 416–430.

[75] Huan Nguyen, Soumyakant Priyadarshan, and R Sekar. 2024. Scalable, sound, and accurate jump table analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis.* 541–552.

[76] Matt Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 27–41.

[77] Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter, Sören van der Wall, and Zhiyuan Zhang. 2025. Transparent Decompilation for Timing Side-Channel Analyses. *arXiv preprint arXiv:2501.04183* (2025).

[78] Vijay Krishna Palepu, Guoqing Xu, and James A Jones. 2013. Improving efficiency of dynamic analysis with dynamic dependence summaries. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 59–69.

[79] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE, 2–14.

[80] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE symposium on security and privacy (SP).* IEEE, 833–851.

[81] Jihee Park, Sungho Lee, Jaemin Hong, and Sukyoung Ryu. 2023. Static Analysis of JNI Programs Via Binary Decompilation. *IEEE Transactions on Software Engineering* (2023).

[82] Jihee Park, Insu Yun, and Sukyoung Ryu. 2025. Bridging the Gap between Real-World and Formal Binary Lifting through Filtered-Simulation. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 898–926.

[83] Lionel Parreaux. 2020. The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl). *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–28.

[84] Fabian Parzefall, Chinmay Deshpande, Felicitas Hetzelt, and Michael Franz. 2024. What you trace is what you get: dynamic stack-layout recovery for binary recompilation. In *Proceedings of the 29th ACM International Conference on*

Architectural Support for Programming Languages and Operating Systems, Volume 2. 1250–1263.

[85] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. StateFormer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 690–702.

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

[87] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

[88] M Pnueli and Micha Sharir. 1981. Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications* (1981), 189–234.

[89] Soumyakant Priyadarshan, Huan Nguyen, and R Sekar. 2023. Accurate disassembly of complex binaries without use of compiler metadata. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 1–18.

[90] Zvonimir Rakamarić and Alan J Hu. 2009. A scalable memory model for low-level code. In *Verification, Model Checking, and Abstract Interpretation: 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings 10*. Springer, 290–304.

[91] Sasko Ristov, Radu Prodan, Marjan Gusev, and Karolj Skala. 2016. Superlinear speedup in HPC systems: Why and when?. In *2016 federated conference on computer science and information systems (fedcsis)*. IEEE, 889–898.

[92] Rodrigo CO Rocha, Dennis Sprokholt, Martin Fink, Redha Gouicem, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. Lasagne: a static binary translator for weak memory model architectures. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 888–902.

[93] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. 2023. Propeller: A profile guided, relinking optimizer for warehouse-scale applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 617–631.

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

[95] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2021. Path-sensitive sparse analysis without path conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 930–943.

[96] Qingkai Shi and Charles Zhang. 2020. Pipelining bottom-up data flow analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 835–847.

[97] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas Reps. 1999. Coping with type casts in C. *ACM SIGSOFT Software Engineering Notes* 24, 6 (1999), 180–198.

[98] Matthew Smithson, Kapil Anand, Aparna Kotha, Khaled Elwazeer, Nathan Giles, and Rajeev Barua. 2010. Binary rewriting without relocation information. *University of Maryland, Tech. Rep* (2010).

[99] Thodoris Sotiropoulos, Stefanos Chaliasos, and Zhendong Su. 2024. API-Driven Program Synthesis for Testing Static Typing Implementations. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 1850–1881.

[100] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. *ACM SIGPLAN Notices* 41, 6 (2006), 387–400.

[101] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. 2021. Demanded abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 282–295.

[102] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. 2024. Interactive Abstract Interpretation with Demanded Summarization. *ACM Transactions on Programming Languages and Systems* 46, 1 (2024), 1–40.

[103] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 254–264.

[104] Zewen Sun, Duanchen Xu, Yiyu Zhang, Yun Qi, Yueyang Wang, Zhiqiang Zuo, Zhaokang Wang, Yue Li, Xuandong Li, Qingda Lu, et al. 2023. BigDataflow: A Distributed Interprocedural Dataflow Analysis Framework. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1431–1443.

[105] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 83–95.

[106] Wensheng Tang, Dejun Dong, Shijie Li, Chengpeng Wang, Peisen Yao, Jinguo Zhou, and Charles Zhang. 2024. Octopus: Scaling Value-Flow Analysis via Parallel Collection of Realizable Path Conditions. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–33.

[107] Dimitrios Vardoulakis and Olin Shivers. 2011. CFA2: A context-free approach to control-flow analysis. *Logical Methods in Computer Science* 7 (2011).

[108] Freek Verbeek, Joshua Bockenek, Zhoulai Fu, and Binoy Ravindran. 2022. Formally Verified Lifting of C-Compiled x86-64 Binaries. In *2022 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[109] Freek Verbeek, Nico Naus, and Binoy Ravindran. 2024. Verifiably Correct Lifting of Position-Independent x86-64 Binaries to Symbolized Assembly. (2024).

[110] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. 2020. Sound C code decompilation for a subset of x86-64 binaries. In *Software Engineering and Formal Methods: 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14–18, 2020, Proceedings 18.* Springer, 247–264.

[111] Jikai Wang and Haoyu Wang. 2024. NativeSummary: Summarizing Native Binary Code for Inter-language Static Analysis of Android Apps. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis.* 971–982.

[112] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. 2022. Odin: on-demand instrumentation with on-the-fly recompilation. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 1010–1024.

[113] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* 1137–1150.

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

[115] Igor Wodiany, Antoniu Pop, and Mikel Luján. 2024. LeanBin: Harnessing Lifting and Recompilation to Debloat Binaries. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering.* 1434–1446.

[116] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2024. ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries. (2024).

[117] Liang Xu, Fangqi Sun, and Zhendong Su. 2009. Constructing precise control flow graphs from binaries. *University of California, Davis, Tech. Rep* (2009), 28.

[118] Zhichen Xu, Thomas Reps, and Barton P Miller. 2001. Typestate checking of machine code. In *Programming Languages and Systems: 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001 Proceedings 10.* Springer, 335–351.

[119] S Bharadwaj Yadavalli and Aaron Smith. 2019. Raising binaries to LLVM IR with MCTOLL (WIP paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems.* 213–218.

[120] Peisen Yao, Jinguo Zhou, Xiao Xiao, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2024. Falcon: A Fused Approach to Path-Sensitive Sparse Data Dependence Analysis. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 567–592.

[121] Wenjia Ye, Bruno C d S Oliveira, and Matías Toro. 2024. Merging Gradual Typing. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 648–676.

[122] Yapeng Ye, Zhuo Zhang, Qingkai Shi, Yousra Aafer, and Xiangyu Zhang. 2022. D-ARM: Disassembling ARM Binaries by Lightweight Superset Instruction Interpretation and Graph Modeling. In *2023 IEEE Symposium on Security and Privacy (SP).* IEEE Computer Society, 728–745.

[123] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. 2022. DEEPDI: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly. (2022).

[124] Bowen Zhang, Wei Chen, Peisen Yao, Chengpeng Wang, Wensheng Tang, and Charles Zhang. 2024. Siro: Empowering Version Compatibility in Intermediate Representations via Program Synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3.* 882–899.

[125] Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation.* 435–446.

[126] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary. In *2021 IEEE Symposium on Security and Privacy (SP).* IEEE, 813–832.

[127] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019. BDA: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–31.

[128] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 427–440.

[129] Peng Zhao and José Nelson Amaral. 2007. Ablego: a function outlining and partial inlining framework. *Software: Practice and Experience* 37, 5 (2007), 465–491.

[130] Anshunkang Zhou, Chengfeng Ye, Heqing Huang, Yuandao Cai, and Charles Zhang. 2024. Plankton: Reconciling Binary Code and Debug Information. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 912–928.

[131] Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. Chianina: an evolving graph system for flow-and context-sensitive analyses of million lines of C code. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 914–929.

Received 2025-10-10; accepted 2026-02-17