Plankton: Reconciling Binary Code and Debug Information

Anshunkang Zhou
azhouad@cse.ust.hk
Hong Kong University of Science and Technology
Hong Kong, China

Chengfeng Ye
cyeaa@cse.ust.hk
Hong Kong University of Science and Technology
Hong Kong, China

Heqing Huang*
heqhuang@cityu.edu.hk
City University of Hong Kong
Hong Kong, China

Yuandao Cai
ycaibb@cse.ust.hk
Hong Kong University of Science and Technology
Hong Kong, China

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

Abstract

Static analysis has been widely used in large-scale software defect detection. Despite recent advances, it is still not practical enough because it requires compilation interference to obtain analyzable code. Directly translating the binary code using a binary lifter mitigates this practicality problem by being non-intrusive to the building system. However, existing binary lifters cannot produce precise enough code for rigorous static analysis even in the presence of the debug information. In this paper, we propose a new binary lifter Plankton together with two new algorithms that can fill the gaps between the low- and high-level code to produce high-quality LLVM intermediate representations (IRs) from binaries with debug information, enabling full-fledged static analysis with minor precision loss. Plankton shows comparable static analysis results with traditional compilation interference solutions, producing only 17.2% differences while being much more practical, outperforming existing lifters by 76.9% on average.

ACM Reference Format:
Anshunkang Zhou, Chengfeng Ye, Heqing Huang, Yuandao Cai, and Charles Zhang. 2024. Plankton: Reconciling Binary Code and Debug Information. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620665.3640382

*Corresponding author

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

ACM ISBN 979-8-4007-0385-0/24/04…$15.00
https://doi.org/10.1145/3620665.3640382

1 Introduction

Static analysis has been widely used in detecting various software defects in C/C++ programs [34–36, 53, 85, 94]. Unfortunately, despite this tremendous progress, one previously-ignored fact is that the requirement of compilation interference is essentially the *last mile* of modern static analysis. Conventional static analysis [3, 13, 16, 33, 37, 82–84] requires access to the program building process to obtain analyzable code, which is realized by interfering with the original compilation configuration with a custom compiler or parser. For example, SVF [84] requires the end-users to use wllvm [20] (a wrapper for `Clang`) to compile the project.

However, compilation interference is not always feasible due to two reasons. First, modern building systems are becoming increasingly complex and diverse [48, 66]. For example, the C++ language has over 20 building systems [23] and 36 compilers [24]. Supporting such a large number of native building systems is extremely difficult and error-prone. For example, Coverity Scan [3], one of the most successful commercial static analysis tools, has been striving to achieve compatibility with various toolchains for over 20 years [31, 76]. Despite tremendous efforts, the success of compilation interference is still not guaranteed. For example, both Coverity Scan [3] and SVF [84] were reported to fail because they could not compile the source code with their custom compilers [2, 4, 6, 12]. Second, the accessibility of the software building procedure is not always guaranteed [31]. For example, the source code and its compilation pipeline are often unavailable for cloud-based code scanning services. The above difficulties severely hinder the deployment of mature static analysis systems to the industrial environment, which we refer to as the *last mile* problem.

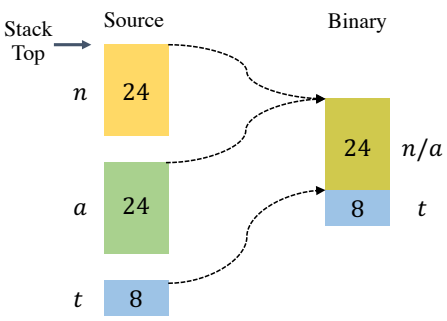
Instead of walking the last mile with insurmountable effort, we can bypass the problem by directly using the end product of the building process — the binaries themselves.

```
1  void foo(int i)
2  {
3    long* v = calloc(2, 8);
4    long  t = 1;
5    {
6      char n[24];
7      n[i] = 'a';
8    }
9    long a[3];
10   a[i] = 2;
11   a[0] += v[0];
12   free(v);
13 }
```

(a) Source code of `foo`.

```
1  ; void foo(int i) ; rdi=i
2    ...                        ;omit
3    call calloc
4    mov  rbx, rax              ;v=calloc
5    mov  [rsp+32], 1           ;t=1
6    mov  [rsp+rdi*8+8],61h     ;n[i]='a'
7    ...                        ;omit
8    mov  [rsp+rdi*8+8],2       ;a[i]=2
9    mov  rax, [rbx]
10   add  [rsp+8], rax          ;a[0]+=v[1]
11   mov  rdi, rbx
12   call free                  ;free(v)
13   ret
```

(b) Assembly code of `foo`.



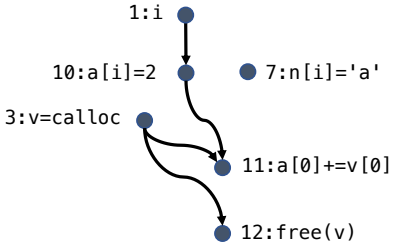
(c) Evolution of the stack frame.

$$\frac{\Gamma \vdash v : ptr(\tau)}{\Gamma \vdash (s = load\ v) : \tau}$$

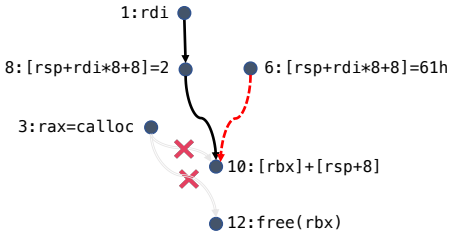
(d) Figure 1a's LOAD typing rule.

$$\frac{\Gamma \vdash v : \alpha}{\Gamma \vdash (s = load\ v) : \alpha}$$

(e) Figure 1b's LOAD typing rule.



(f) DDG of Figure 1a.



(g) DDG of Figure 1b.

Figure 1. Dots in data-dependence graph (DDG) are marked by code lines. In Figure 1d and Figure 1e, Γ maps variables to their types, $ptr(\tau)$ is the type of ordinary pointers to τ, α is a uni-type to describe the untyped assembly language.

Compared with the conventional solution, the binary-based one obtains code for analysis by binary lifting [5, 9, 11, 63], which is non-intrusive to the software building system and thus is unlikely to fail in real-world scenarios. Therefore, the remaining problem is: *Can we achieve similar static analysis results on the lifted code as the compiled source code?*

1.1 Challenges

Rigorous static analysis requires considerable source-level information to be present in the target code to reason about their semantics [78]. Existing state-of-the-art static analysis mainly leverages underlying assumptions about two program entities to develop their algorithms, i.e., variable boundaries [60] and type information [68]. However, the binary code is not naturally accompanied by such information, and recovering it from the plain binary is still an undecidable problem [52, 89–91], which could greatly affect the precision of static analysis. We use Figure 1 to briefly introduce how the loss of variable boundaries and type information in binary code affects the construction of data-dependence graph (DDG), one of the most basic static analyses.

Variable boundaries. Unlike the source code, where the memory is separated into disjoint variables [61], the assembly code flattens the memory space by allowing memory access across variables [60] and possible overlapping objects [54]. For example, the memory write regarding `n[i]` at Line 7 and the memory read from `a[0]` at Line 11 in Figure 1a do not have any data dependences because every local variable is associated with a distinct memory block (left-hand side of Figure 1c). Therefore, the DDG has no edge between those two lines in Figure 1f. However, the compiler places `n` and `a` into the same stack slot (`[rsp+8]`) in the binary code since they have non-overlapping scopes (right-hand side of Figure 1c), resulting in a bogus dependence between Line 6 and Line 10 in Figure 1b. Therefore, there is an extra edge between them in Figure 1g.

Type information. Binary code totally discards high-level type information and allows operations between arbitrary typed values. For example, the typing rule for the memory read in source code (Figure 1d) requires the operand v to be a pointer to type τ, and the type of the loaded value s is also τ. In contrast, the assembly code (Figure 1e) uses a uni-type α for all operands. Existing static analysis tools exploit the type information for efficiency [79], and losing it might lead to broken results. For example, the pointer returned by `calloc` at Line 3 in Figure 1a has data dependences with Line 11 and Line 12. In Figure 1b, the memory allocated by `calloc` is returned and manipulated through two integer-typed registers `rax` and `rbx` (Line 9 and Line 12)), which might result in broken data-flows in some static analyzers (e.g., SVF) since they ignore pointer values that behave like an integer. Therefore, the constructed DDG in Figure 1g lacks two edges starting

from Line 3 to its reference sites and will even result in a false alarm of memory leak bug because the `calloc` does not reach a `free` function. Moreover, heap objects cannot be modeled properly without correct type information at the call sites of memory allocation functions [55].

Fortunately, while the plain binary is not a desired substitution for the compiled source code, we found that in a scenario like software static analysis, the *debug information* [45, 62, 67] is also indispensable because it is extremely useful for debugging [14, 21], crash diagnosis [43], and profiling [17]. For example, modern operating systems all provide debug symbol packages [18, 22], and the latest Firefox on MacOS is built with embedded debug information [62]. In a small study conducted by us that involves 100 famous C/C++ projects from Linux and GitHub (projects with the most stars), we found that 100% of them support emitting debug information in their binary output. Therefore, a more feasible solution would be considering the debug information during binary lifting, which maps the binary to the source code and can help solve numerous undecidable reverse engineering problems [73].

Despite its usefulness, the debug information only provides a subset of source-level information since compilers cannot backward map all binary code to the source [28, 62, 74, 95], which still constrains the precision of lifted code and static analysis because variable and type information are still missing in some code fragments (illustrated in § 2).

1.2 Our Technique

In this paper, we propose a new binary lifter Plankton together with two new algorithms to produce high-quality LLVM intermediate representations (IRs) from binaries with debug information, enabling full-fledged static analysis with minor precision loss. Our key insight is that code fragments with data-dependences are also dependent in terms of their source-level properties, which allows us to fill in the blankets in the binary-to-source mapping with the incomplete debug information by additional static analysis and code transformations. More specifically, we first propose a stack disambiguation algorithm to reconstruct connections between variable entities and stack references in the lifted code. Our intuition is that in source code, references to stack slots can be mapped to disjoint memory blocks represented by top-level variables visible at specific program points, which is essentially the scope information. Therefore, our algorithm computes a scope for each variable by extending existing ones denoted by the debug information. It leverages data-dependences between instructions to group stack memory references and uses structural invariants to form well-structured variable scopes. Second, we propose a type enforcement algorithm to recover proper types for all values inside the lifted code. Our intuition is that all operations in a strongly-typed language such as C/C++ obey strict typing rules, meaning that values with improper types must be explicitly converted to satisfy those rules to keep the code's validity. Therefore, the usage of type conversions indicates the degree of type correctness because the code with more correctly typed values needs fewer conversions. Starting from a portion of value types, our algorithm consistently eliminates type conversions with semantic-preserving code transformations and propagates the results to all code fragments to find a fixed point that can maximize the type correctness level.

We have implemented Plankton in C/C++ on top of the LLVM compiler infrastructure [58]. We evaluated it with real-world programs using two state-of-the-art static analysis tools, SVF [84] and Pinpoint [83]. Experimental results show that Plankton is able to fully utilize source-level static analysis techniques by showing comparable bug-finding results with traditional compilation interference solutions, producing only 17.2% differences while being much more practical, outperforming existing lifters by 76.9% on average. Our further experiments also prove that most of the remaining result differences are caused by the inherent "randomness" of the static analysis, showing that Plankton is as effective as the traditional compilation interference method for real-world usage.

In summary, this paper makes the following contributions:

- We propose two new algorithms to produce precise LLVM IRs from binaries with debug information to enable full-fledged static analysis.
- We implement a new binary lifter Plankton that incorporates the proposed algorithms. We also spent significant engineering efforts handling complicated cases in parsing debug information and lifting.
- We conduct experiments to show that Plankton has sufficiently good scalability and precision, producing only 17.2% differences compared with the compiled source code while being much more practical, outperforming existing lifters by 76.9% on average.

2 Motivation

2.1 Motivating Example

Translating the binary product of the software into analyzable code representations has greatly potential to enable a more practical static analysis procedure because it is non-intrusive to the building system. However, applying static analysis designed for high-level languages to the low-level assembly code could cause great precision loss because of the huge gaps between these two languages.

We use a motivating example to show the two core differences between the binary and the source code that could affect the static analysis. Figure 2a shows the source code of function `foo`, and Figure 2b shows the corresponding `X86_64` assembly code obtained by compiling `foo` with "`Clang -O1 -g`". We use the `sc.qitem=self->qitem` statement at Line 7

```
1  int foo(struct Cmd* self)
2  {
3    struct Cmd*  t = init();
4    struct Item* item = NULL;
5    struct Ctx   sc = {0};
6    char*        cause = NULL;
7    sc.qitem = self->qitem;
8    sc.argv  = t->argv;
9
10   args_proc(&sc.argc,&sc.argv);
11   if (!get_ctx(&sc, &cause))
12   {
13     char msg[10];
14     msg[2] = ':';
15     cmd_proc(sc.argv);
16   }
17   else
18   {
19     struct LogClass log;
20     dump_log(&log, log.msg);
21     cmd_proc(sc.argv);
22   }
23 }
```

(a) Source code

```
B0:
  ...
  lea  r14, [rsp+32] ; &sc.argc
  mov  rax, [rdi+8]  ; self->qitem
  mov  [rsp+48], rax ; sc.qitem
  mov  rax, [r15]    ; t->argv
  mov  [rsp+24], rax ; &sc.argv
  lea  rsi, [rsp+24] ; &sc.argv
  mov  rdi, r14      ; &sc.argc
  call args_proc
  lea  rdi, [rsp+16] ; &sc
  ...
B1:
  lea  rsi, [rsp+68] ; log.msg
  lea  rdi, [rsp+64] ; &log
  call dump_log
  mov  rdi, [rsp+24] ; sc.argv
  call cmd_proc
  ...
B2:
  mov  [rsp+66], 3Ah ; msg[2]
  mov  rdi, [rsp+24] ; sc.argv
  call cmd_proc
```

(b) Assembly code

```
define i32 @foo(%Cmd* %self) {
B0:
  %msg = alloca [10 x i8]
  %log = alloca %LogClass
  %sc = alloca %Ctx
  %cause = alloca i8*
  %field.1 = getelementptr %sc, 0, 2
  %field.2 = getelementptr %sc, 0, 1
  %field.3 = getelementptr %sc, 0, 5
  ...
  %t21 = getelementptr %self, 0, 1
  %t22 = load %Item** %t21
  store %t22, %field.3     ; sc.qitem
  %t23 = getelementptr %t19, 0, 0
  %t24 = load i8** %t23    ; t.argv
  store %t24, %field.2     ; sc.argv
  ...
  %t25 = call @get_ctx(%sc, %cause)
  ...
B2:
  %t33 = load i8** %field.2
  call void @cmd_proc(%t33)
}
```

(c) LLVM IR produced by Plankton

Figure 2. A motivating example that shows the original source code of function `foo`, the compiled assembly code, and the translation result of Plankton. All three pieces of code are simplified for readability. Colored lines show corresponding code statements (one source line might be mapped to multiple assembly instructions). The annotations aim at easier understanding and do not represent the actual debug information to source code mapping.

in Figure 2a to illustrate the problem, which is compiled to the `mov [rsp+48],rax` instruction at Line 5 in Figure 2b.

Variable boundaries. In Figure 2a, `sc` has an independent memory block so that `sc.qitem` can only be accessed when there is a reference to `sc`. In contrast, in Figure 2b, all local variables are placed in a flat stack memory and are accessed by relative offset to the `rsp` register. Therefore, the stack slot `[rsp+48]`, which corresponds to `sc.qitem`, might be accessed whenever there are references relative to the `rsp`, which forces static analysis to reason about bogus dependences relations and might cause precision loss.

Type information. In Figure 2a, the statement follows strict typing rules, and values have proper types, i.e., it writes an `Item` typed pointer to `sc.qitem`, which is also an `Item` typed pointer. However, in Figure 2b, the value held by `rax` register and `[rsp+48]` can have arbitrary types, which could mislead static analysis to produce wrong results.

Even the debug information is still insufficient to fill the gap between binary and source code since compilers cannot backward map all binary code to the source [28, 62, 95]. For example, it tells that the `[rsp+16]` at Line 11 in Figure 2b is the `sc` variable in the source code but does not map `[rsp+24]` at Line 17 to any variables.

2.2 Limitations of Existing Approaches

Existing binary lifters [5, 8, 9, 11, 42] all suffer from numerous limitations that make them unable to support rigorous static analysis. First, lifters such as McSema [5] output the emulation-style IR that faithfully emulates the machine execution, which is too low-level to support source-level static analysis [63]. Second, lifters such as RetDec [11] only directly apply knowledge inside the debug information to the lifted code without being aware of its incompleteness and thus suffer from significant precision loss.

Moreover, we found that even the most advanced decompilers, such as IDA Pro [1] and Ghidra [15], still cannot precisely translate binaries with debug information because their algorithms usually depend on predefined rules or patterns. As a result, in the decompiled code of IDA Pro, stack references at Line 3 and Line 8 in Figure 2b are declared as independent values instead of `sc`'s fields. Additionally, significant manual efforts are needed to re-compile outputs of those decompilers into analyzable code representations and are not always feasible for large programs [65].

```
1  B0:
2    %argv = getelementptr %sc, 0, 1
3  ; mov  rax, [r15]
4    %41 = load i64* @r15
5    %42 = inttoptr i64 %41 to %Cmd*
6    %43 = ptrtoint %Cmd* %42 to i64
7    %44 = inttoptr i64 %43 to i64*
8    %45 = load i64* %44
9    store i64 %45, i64* @rax
10 ; mov  [rsp+24], rax
11   %47 = load i64* @rax
12   %48 = inttoptr i64 %47 to i8*
13   store i8* %48, i8** %argv
```

(a) Vanilla LLVM IR.

```
B0:
  %argv = getelementptr %sc, 0, 1
; mov  rax, [r15]
  %47 = inttoptr i64 %36 to %Cmd*
  %48 = bitcast %Cmd* %47 to i64*
  %49 = load i64* %48
; mov  [rsp+24], rax
  %52 = inttoptr i64 %49 to i8*
  store i8* %52, i8** %argv
```

(b) After optimizations.

```
B0:
  %47 = getelementptr %30, 0, 0
  %48 = load i8** %47
  %49 = ptrtoint i8* %48 to i64
  %52 = inttoptr i64 %49 to i8*
  store i8* %52, i8** %argv
```

(c) After 1st transformations.

```
B0:
  %47 = getelementptr %30, 0, 0
  %48 = load i8** %47
  store i8* %48, i8** %argv
```

(d) After 2nd transformations.

Figure 3. The evolution of the LLVM IR translated from Line 6 and Line 7 in Figure 2b. Some instructions are omitted.

2.3 Our Technique

Insight. Our key insight to overcome the above limitations is that code fragments with data-dependences are also dependent in terms of their source-level properties, which allows us to fill in the blankets in the binary-to-source mapping with the incomplete debug information. Based on this insight, we propose two new algorithms to better resolve the two most critical problems in binary lifting.

2.3.1 Variable Boundaries. The flattened memory in binary code should be separated into disjoint variables to enable static analysis. However, existing lifters only map a portion of stack references to variable entities due to the incompleteness of debug information. For example, the stack reference at Line 11 in Figure 2b is mapped to the `sc` variable, but `[rsp+24]` at Line 17 does not have any mappings.

Intuition. Our intuition is that the *live range* [32] of identified variables allows us to connect binary-level memory references to source-level variables. For example, with complete scope information of `sc`, we can tell that `[rsp+24]` at Line 17 actually is a field of the `sc` variable.

Therefore, our first algorithm infers a scope for each local variable and uses such information to recover precise variable information. More specifically, it extracts a set of constraints by exploiting several scope invariants (e.g., overlap variables have disjoint scopes) and data-dependences between memory references through static analysis, which, when taken together with the debug information, allows us to solve them uniformly to obtain the final scope of each variable. For example, our algorithm collects two data-flow facts for `sc` referred at Line 11 in Figure 2b. First, the accessed value is defined by Line 7 by moving `rax` to `[rsp+24]`. Second, Line 22 loads the same value from `[rsp+24]` since there is no redefinition in the path between these two lines. They indicate that `[rsp+24]` at these three lines actually refer to the same value and thus belong to the same local variable. Then, PLANKTON combines the obtained facts with a scope invariant that the variable scope should be continuous and draws a scope for `[rsp+24]` (`sc`) according to the CFG, which covers `sc` variable at Line 11. Moreover, the byte size of `sc` (`[rsp+16]`) in the binary code is 40, which means that the `[rsp+24]` must be the field of `sc` instead of an independent variable since $24 - 16 < 40$ and different variables must not overlap. Therefore, the scope of `sc` can be extended to Line 7 - Line 22 after the above analysis.

2.3.2 Type Information. All values inside the lifted IR should be assigned with correct types to facilitate static analysis. However, existing lifters still could leave a large number of values with improper types because debug information only maps a small portion of values to their correct types. For example, `r15` is marked as `Cmd*` type at Line 6 in Figure 2b, but `rax` at Line 7 is untyped.

Intuition. Our intuition is that all operations in a strongly-typed language obey strict typing rules [58], meaning that values with improper types must be explicitly converted to satisfy those rules. Therefore, if we can eliminate such type conversions while keeping the validity of the LLVM IR, then all operations can be assigned with proper types, which further indicates that the usage of type conversions reflects the degree of type correctness because the code with more correctly typed values needs fewer conversions.

Based on this intuition, our second algorithm leverages semantic-preserving code transformations to consistently eliminate type conversions and assign correct value types. For example, Figure 3a shows the vanilla LLVM IR directly translated from the assembly instructions at Line 6 and Line 7 in Figure 2b. Our algorithm first leverages code optimizations to remove redundant operations in Figure 3a and make dependences between values more explicit (Figure 3b). Then, it transforms the **inttoptr** and **bitcast** instructions into a **getelementptr** and modifies the type of the **load** instruction (Figure 3c). The loaded value is then translated to the original type by inserting a **ptrtoint** to keep the validity of

code. After this step, we not only assign the correct type to the **load** instruction but also "propagate" type hints forward to the next instruction. Further applying transformations gives Figure 3d, which has no type conversions.

Figure 2c shows the translation result of Plankton, which is very similar to the one compiled from the source code.

3 Stack Disambiguation

In this section, we will introduce how Plankton tackles the most challenging part of recovering source-level variables in the lifted code, i.e., separating the stack memory into disjoint blocks. For global memory, Plankton adopts similar algorithms with existing methods [30], details are omitted.

Plankton first obtains an initial result by analyzing operations involving the stack pointer (e.g., `rsp` in `X86_64`) and transforms them into references to stack slots [1, 11, 15, 29, 49]. At the same time, the debug information is encoded by annotating some stack slots with types and names.

Example 3.1. In Figure 2b, stack operations at Line 11 and Line 17 are transformed into references to stack slots (LLVM **alloca**) but only Line 11 is further mapped to the `sc` variable.

Then, it infers a live range [32] (scope) for each identified variable with Algorithm 1. It infers a must (§ 3.1) and a may (§ 3.2) scope for *interference* stack slots (Line 2), which could be variables with non-overlapping scopes or belong to the same variable or its fields. We now define *interference*:

Definition 3.1. *Two stack slots x_1 and x_2 interfere (overlap) if they have the same stack memory offset or one slot occupies the middle of the other.*

Example 3.2. `[rsp+16]` at Line 11 and `[rsp+24]` at Line 17 in Figure 2b are a pair of stack slots with interference because `[rsp+16]` is mapped to a variable with size 40 (`sc`) and is larger than their distance inside the memory.

3.1 Must Scope Inference

The must scope is a precise but under-approximated live range for a specific stack slot v (Line 4 - Line 18). The set φ_{must} represents the must scope for v, which is essentially a set of instructions where v is visible. Initially, φ_{must} only contains all direct use sites of v since a variable must be visible where it is used (Line 4).

The algorithm first collects the set of program points (σ_u) where v must be visible with data-flow analysis that concerns how values are defined and used (Line 6). More specifically, if a value is stored into v, we add all instructions that load the stored value in v to σ_u. Similarly, if a value is loaded from v, we find all instructions that store the loaded value to v and mark them as must live sites of v. Moreover, we also track values loaded and stored through registers by performing a lightweight intra-procedural pointer analysis to find alias relations between stack slots and registers.

Algorithm 1: Scope Inference

Input: *Function F after recovering stack slots*
Output : *Updated Function F with mapped variables*

1 **Function** `StackDisambiguation(`F`)`:
2 $S \leftarrow$ *stack slots that have overlapping memory*
3 **foreach** v *in* S **do** ▷ *collect scope constraints*
4 $\varphi_{must} \leftarrow$ *get all direct use sites of* v ▷ *must scope*
5 **foreach** u *in* φ_{must} **do** ▷ *collect data-flow facts*
6 $\sigma_u \leftarrow$ *data-flow facts regarding* u
7 $\varphi_{must} \leftarrow \varphi_{must} \cup \sigma_u$
8 $\varphi_w \leftarrow \varphi_{must}$ ▷ *worklist for applying invariants*
9 **while** $\varphi_w \neq \emptyset$ **do**
10 $u \leftarrow \varphi_w.popBack()$
11 $\varphi'_w \leftarrow applyInvs(u, \varphi_{must})$ ▷ *apply invariants*
12 $\varphi_{must} \leftarrow \varphi_{must} \cup \varphi'_w$
13 $\varphi_w \leftarrow \varphi_w \cup \varphi'_w / \varphi_{must}$
14 **foreach** v *in* S **do** ▷ *merge stack slots that belong to the same variable*
15 $S_v \leftarrow$ *stack slots overlap with* v
16 **foreach** v' *in* S_v **do**
17 **if** $\varphi_{must}(v) \cap \varphi_{must}(v') \neq \emptyset$ **then** ▷ *both live*
18 *Collapse*(v, v')
19 $Q_f \leftarrow$ *reverse post order of* F ▷ *calculate may scope*
20 $Q_b \leftarrow$ *post order of* F
21 **foreach** I *in* F **do**
22 $USE[I], KILL[I], OVER[I] \leftarrow init(\varphi_{must})$ ▷ *init sets*
23 **while** *changes to any* OUT *or* IN *occur* **do** ▷ *backward*
24 **foreach** *instruction* I *in* Q_f **do**
25 $OUT[I] \leftarrow \bigcup_{S \in succ[I]}(IN[S]) - OVER[I]$
26 $IN[I] \leftarrow USE[I] \cup (OUT[I] - KILL[I])$
27 **while** *changes to any* OUT *or* IN *occur* **do** ▷ *forward*
28 **foreach** *instruction* I *in* Q_b **do**
29 $IN[I] \leftarrow \bigcup_{P \in pred[I]}(OUT[P]) - OVER[I]$
30 $OUT[I] \leftarrow USE[I] \cup (IN[I] - KILL[I])$

Example 3.3. The instruction at Line 3 in Figure 2b stores the address of `[rsp+32]` to `r14`; therefore, reading memory pointed to by `r14` is equivalent to reading from `[rsp+32]` before `r14` is assigned with another value, and the corresponding instructions will be added to σ_u.

However, program points collected by the above data-flow analysis could spread across the whole function and the result ($\varphi_{must} \cup \sigma_u$) may not form a well-defined continuous variable scope. Therefore, Plankton further leverages a set of scope invariants to refine and expand the result obtained from data-flow facts (Line 10 - Line 13). The intuition is that in a high-level language, the scope of local variables should respect certain control-flow structures [56], which applies to the compiled binary code. For example, C/C++ programs usually have a naturally scoped structure, where the lifetime

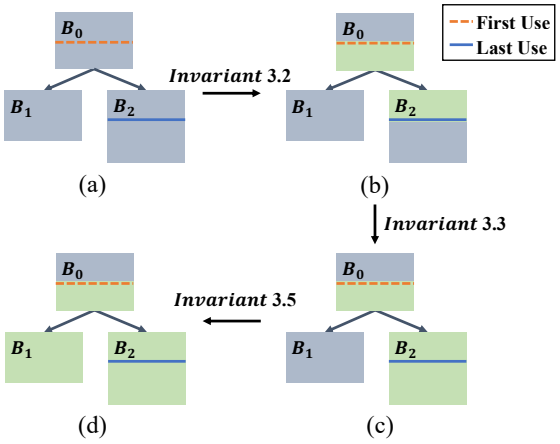


Figure 4. An illustration of scope invariants.

of data is bounded by curly brackets. The basic principle of variable scope is that each scope has to be continuous, and two different scopes are either the same, nested, or disjoint. More formally:

Invariant 3.1. *Let variables x_1 and x_2 have scopes σ_1 and σ_2. Then, both σ_1 and σ_2 are continuous, and either σ_1 and σ_2 are the same, disjoint or nested, but not overlapping:*

$$(\sigma_1 = \sigma_2) \vee (\sigma_1 \cap \sigma_2 = \emptyset) \vee ((\sigma_1 \subset \sigma_2) \vee (\sigma_2 \subset \sigma_1))$$

The implication of Invariant 3.1 is that given some program points that a variable must live, we can infer its complete live range using additional structure analysis, resulting in several derived invariants.

We use Figure 4 to show how scope invariants are used to infer variable scopes. Two different kinds of lines represent the first and the last use site of a stack slot, which are also identified as must live program points by the previous data-flow analysis. Since the scope for each variable is continuous, the following invariant holds for variables:

Invariant 3.2. *If a variable x is visible at two reachable program points p_i and p_j, then it is also visible at all program points between p_i and p_j, denoted as $\langle p_i, p_{i+1}, ..., p_j \rangle$.*

Example 3.4. By applying Invariant 3.2 to Figure 4a, we can obtain Figure 4b, which includes all program points between the start and end use sites into the scope of the stack slot.

In a CFG, each basic block can be viewed as an independent scope bounded by curly brackets. Therefore, if a variable scope contains more than one basic block, then the scope should fully cover each block. Otherwise, the variable scope will overlap with the scope of individual basic blocks. Some compilers further optimize the use of stack slots by taking their first uses as the lifetime start [19]. However, it is hard to determine the end of lifetime statically, so compilers still use the end of declaration scope as the end of stack slots' lifetime [50]. Therefore, we have the following invariant:

Invariant 3.3. *If a variable x is visible at more than one basic block, its scope fully covers all blocks except the starting one.*

Example 3.5. In Figure 4b, the variable scope partially covers B_0 and B_2. Therefore, it can be extended to Figure 4c by including the whole B_2 in the scope.

The start and end program points of the variable scope should have dominance relations:

Invariant 3.4. *The end/start program points of the scope should (post-) dominate the start/end program points.*

, which can further derive the following invariant:

Invariant 3.5. *If a variable x is visible at some program points inside block B and one of B's successors (predecessors) B'. Then x must live at B's other successors (predecessors) that are unreachable from B'.*

Example 3.6. In Figure 4c, the end of the scope in B_2 does not post-dominate the start in B_0. Therefore, B_1 should also be included in the scope, which gives us Figure 4d.

Additionally, all aggregate typed variables should respect the following invariant:

Invariant 3.6. *A variable x is visible at all program points where any of x's fields are visible.*

A worklist-based algorithm iteratively applies the above invariants to all live sites of v (Line 11). Initially, the worklist φ_w only contains all use sites obtained from data-flow facts φ_{must}. In each iteration, function *applyInvs* analyzes one instruction u against all the other live sites of v using invariants 3.1 - 3.6, and the returned set φ'_w contains extended live sites of v (Line 11). Newly discovered live sites (φ'_w) are added to the worklist and φ_{must} (Line 12 and Line 13). The algorithm continues until φ_w becomes empty. φ_{must} contains all must live sites of v.

The obtained must live scopes for stack slots are further used as scope constraints to reason about relations between stack slots and top-level variables. For each stack slot v, stack slots that overlap with it are identified as S_v (Line 15). If a program point exists where v and one of its overlapping counterpart v' both live, v and v' are collapsed using a union-find data structure (Line 18) since they must belong to the same variable. All collapsed slots are extended to share the same must scope and are made to belong to the same variable when updating the function.

3.2 May Scope Inference

Since the must scope is an under-approximation, it might miss some instructions where the variable is visible. Moreover, such incomplete scope information might break data-dependence relations between memory references and hurt the soundness of static analysis.

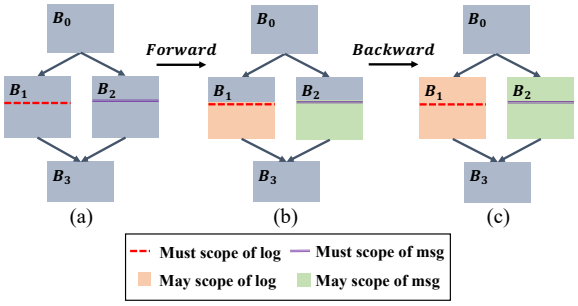


Figure 5. May scope inference.

Example 3.7. Consider the stack slot accessed (`[rsp+64]`) at Line 15 in Figure 2b, which is identified as the variable `log`. The must scope of `log` only contains a single instruction at Line 15 because it does not have data-dependences with others, which results in missing the field access at Line 14 (`[rsp+68]`).

Therefore, in order to produce a sound data-flow recovery result, PLANKTON further calculates a may scope for stack slots in S (Line 19 - Line 30).

The may scope is calculated by an augmented liveness analysis that considers possible stack slot overlapping during forward and backward data-flow analysis. Traditional liveness analysis [69] only performs backward analysis, which propagates the use site information to previous instructions. However, there may not be explicit uses after live sites of a stack slot. For example, the field access at Line 14 in Figure 2b could be placed before or after Line 15.

Five sets are maintained for each instruction: *OUT*, *IN*, *USE*, *KILL*, and *OVER*. Initially, *OUT* and *IN* are empty. *USE* contains stack slots that must live at current instruction, which comes from the result of previous must scope inference (Line 22). The *OVER* and *KILL* sets are used to minimize possible precision loss (merging unrelated stack slots) introduced by the may analysis. The idea here is to leverage stack slots that have contradicted visibility to filter out inaccurate may scope results, which is defined below:

Definition 3.2. *(Visibility Contradiction). Stack slots x_1 and x_2 have visibility contradiction if they interfere in the memory and are independent variables in the debug information.*

In other words, two variables with disjoint scopes in the source code could occupy the same memory location and cannot be visible simultaneously in the binary code.

The *KILL* set of one instruction contains stack slots that must not be visible given the ones in *USE*. The *OVER* set is used at some merge points in the control-flow graph. When multiple sets are merged into a single one, we check if there exist stack slots that cannot be visible at the same time. If so, we will abandon both of them.

Example 3.8. Stack slots corresponding to `log` and `msg` in Figure 2b must not be visible together because they are independent variables and occupy the same memory location `[rsp+64]`. Therefore, the *KILL* set at Line 15 in Figure 2b contains `msg`, which prevents the may scope of `msg` from including `log`'s scope. Similarly, the *OVER* set after Line 23 contains both `log` and `msg`, which rules out the false scope starting from Line 23 for both variables.

Similar to traditional data-flow analysis, we define transfer functions for backward and forward data-flow propagation as illustrated in Line 25 to Line 26 and Line 29 to Line 30, respectively. In Algorithm 1, two worklist-based processes iteratively apply transfer functions on instructions along the control-flows. The algorithm keeps updating these sets until it sees the fixed point (Line 19 - Line 30). The *IN* set of each instruction is taken as the set of may live slots.

Example 3.9. In Figure 2b, after the must scope inference, the stack slot `[rsp+64]` represents the `log` at Line 15 while referring to the `msg` at Line 21. The stack slot `[rsp+68]` at Line 14 does not map to any variables. Figure 5 illustrates the may scope inference process for `log` and `msg`. Two different kinds of lines represent their must live sites after the must scope inference. The colored areas represent the extended live ranges obtained by the may scope inference. Initially, the *USE* sets of Line 15 and Line 21 contain `log` and `msg`, respectively (Figure 5a). Figure 5b shows that the forward analysis only propagates the *USE* sets to program points after the must live sites. The propagation stops at the beginning of B_3 since the two variables overlap and cannot be visible simultaneously in B_3. Therefore, their may scopes are extended to program points before B_3 and after must live sites. Similarly, Figure 5c shows that the backward analysis extends their may scopes before must live sites and after B_0. After the backword and forward analysis, only B_1 and B_2 are marked as the may scopes for `log` and `msg`, respectively.

3.2.1 Applying Scopes.

Finally, PLANKTON transforms the function according to inferred scopes. The must scope is first used for reconstructing the mapping by finding if specific references belong to any scopes of source-level variables. Then, PLANKTON further leverages the may scope information to obtain a sound translation result by merging unmapped stack slots to the mapped ones. Since the may scope is an over-approximation, we could wrongly merge independent stack slots into one variable entity. We found that the may scope can benefit static analysis according to the evaluation in § 5.4.2, meaning that although the may analysis cannot entirely avoid precision loss, it brings more true positive results than false positives. It is because a precise static analysis is able to eliminate the effect of fake dependences by combining different levels of sensitivity [71].

4 Type Enforcement

In this section, we introduce the type enforcement algorithm. The initial LLVM IR is first annotated with various type

Promotion Rules

$$\frac{\begin{array}{c} v = \textbf{ptrtoint}\ \tau*\ v_p\ \textbf{to}\ isz \\ (\text{isAgg}(\tau) \wedge (v_k = v \vee v_j = v)) \Rightarrow [\![v_i = \textbf{add}\ isz\ v_j,\ v_k]\!]_n \end{array}}{\begin{array}{c} v' = \textbf{getelementptr}\ \tau*\ v_p,\ \textbf{i32}\ [\![v_{idx}]\!]_m \\ v = \textbf{ptrtoint}\ \text{typeOf}(v')\ v'\ \textbf{to}\ isz \end{array}} \quad \texttt{(multiAdd)}$$

Demotion Rules

$$\frac{\exists k, v_k = \textbf{inttoptr}\ isz\ v_p\ \textbf{to}\ \tau*;\ v = \textbf{phi}\ \tau*\ [\![[v_i,\ b_i]]\!]_n}{\begin{array}{c} [\![v_i' = typeConv(v_i, isz)]\!]_n \\ v_t = \textbf{phi}\ isz\ [\![[v_i',\ b_i]]\!]_n \\ v = \textbf{inttoptr}\ isz\ v_t\ \textbf{to}\ \tau* \end{array}} \quad \texttt{(intPhi)}$$

Figure 6. Example promotion and demotion rules.

information obtained from the debug information including variables, function parameters, etc.

Then, PLANKTON iteratively applies both LLVM native optimizations (e.g., `InstCombine`) and a set of custom peephole transformation rules to the initial IR. We instantiate a type lattice to organize all LLVM types. The top ($\top$) of the lattice is the least precise type, representing the any type. The bottom ($\bot$) of the type lattice is the most precise type. For example, all values in the assembly code initially have the integer type, which is the most imprecise one, and a more precise one could be a pointer type. In the initial program, apart from values without type annotation whose types are under-precise, some value types can also be over-precise. For example, inlined functions could be optimized to use the pointee value directly, however, the debug information might still mark the variable as the original pointer type, which makes the variable's type "more precise" than its actual type.

4.1 Transformation Rules

To tackle the above problems, we propose two kinds of transformations to minimize type conversions: promotion and demotion. Promotion moves down the type lattice, finding increasingly precise types for values, while demotion moves up the type lattice to find decreasingly less precise types. They also have an inverse correlation in that one can always undo the effect of the other. Apart from assigning correct types to values, those rules also propagate type hints forward through the control- and data-flows because type conversions are passed on to all use sites of the values.

Figure 6 shows two example transformation rules [1]. Each rule has two parts and are split by a single line, where the upper part represents the code pattern we are trying to catch, and the lower part is the semantically equivalent transformation result. The bold font represents LLVM instructions. typeof(v) returns the type of value v. typeConv(v, τ) inserts type conversions to convert v to type τ. replaceUse(v_1, v_2) will replace all uses of v_1 in the LLVM module with v_2. isAgg(τ) returns true if τ is an aggregate type. typeOfPointee(v) returns the pointee type of the pointer value v. Notation $[\![op_i]\!]_n$ means repeat op_i for n times like $op_0, op_1, ..., op_n$. We also define a partial order for each pair of types, denoted as $\tau_1 > \tau_2$, which means τ_1 is more precise than τ_2 on the type lattice.

`multiAdd` is a general transformation rule for converting pointer arithmetic with incorrect types into the correct ones [58]. Given a **ptrtoint** instruction and a sequence of **add** instructions, PLANKTON transforms them into a **getelementptr** instruction by finding proper indices into sub-elements of aggregate types. Operands used in each **add** instruction could be either immediate value or variable. The immediate offset can index types with variable-length sub-elements (e.g., structure), while variable offset can only be used in types with fixed-length sub-elements (e.g., array). The demotion rules are just a set of "inverse" rules for the promotion. `intPhi` demotes the pointer type used in the **phi** instruction to the integer type by adding type conversions.

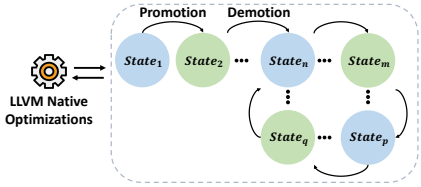
4.2 Fixed point Algorithm

As illustrated in Figure 7a, PLANKTON applies both LLVM native optimizations and peephole transformation rules to the IR. $State_i$ represents the number of type conversions. Promotion and demotion are performed back-and-forth, and reach their only fixed points (the number of conversions does not change) in each iteration. The algorithm also finds a global fixed point between promotion and demotion. Since LLVM native optimizations are not guaranteed to reach a fixed point, our algorithm only iterates between optimizations and peephole transformations for a fixed number of times (3 in our implementation to balance the effectiveness and efficiency).

Termination of the algorithm. Both promotion and demotion are guaranteed to terminate at $State_i$ because any of them alone is monotonic according to the type lattice, and the number of type conversions bounds the search. All state transitions are organized into a graph, and PLANKTON decides whether a global fixed point is reached by detecting cycles on-the-fly. The fixed point could contain multiple states because conflict types might exist for the same value. A typical example is the usage of union type inside the code, which can be cast to different types according to the context. As shown in Figure 7a, the global fixed point can have 2 ($State_p$ equals to $State_n$), 4, or $2n$ states. PLANKTON will choose the state with the fewest type conversions as the final state. Therefore, the global fixed point could be dynamic and guaranteed to be reached since the number of type conversions is bounded.

Example 4.1. Figure 7b shows the LLVM IR after promotion. Since it still contains a **ptrtoint**, PLANKTON performs another round of demotion, resulting the IR in Figure 7c. It can be seen that the states of Figure 7b and Figure 7c form a

[1] Due to page limitation, a formal description of the syntax and more rules can be found in the supplement material



(a) Overview

```
B0:
  %5 = getelementptr i64* %4, 1
B1:
  %20 = getelementptr i64* %15, 2
B2:
  %60 = phi i64* [%5, %B0], [%20,
    %B1]
  %61 = ptrtoint i64* %60 to i64
  sub i64 %61, i64 %54
```

(b) LLVM IR After promotion.

```
B0:
  %5 = getelementptr i64* %4, 1
  %7 = ptrtoint i64* %5 to i64
B1:
  %20 = getelementptr i64* %15, 2
  %21 = ptrtoint i65* %20 to i64
B2:
  %60 = phi i64 [%7, %b_0], [%21,
    %b_2]
  sub i64 %60, i64 %54
```

(c) LLVM IR after demotion.

Figure 7. An illustration of back-and-forth transformations. Some instructions are omitted for simplicity.

cycle in the graph. Therefore, PLANKTON identifies them as a global fixed point that involves 2 states and takes Figure 7b as the final result since it contains fewer conversions.

5 Evaluation

In this section, we aim to systematically evaluate the effectiveness (§ 5.1) and efficiency (§ 5.3) of PLANKTON on large-scale benchmarks. Additionally, we conduct ablation studies to illustrate the importance of PLANKTON's key components (§ 5.4).

Environment. All experiments are performed on an Intel Xeon(R) computer with an E5-1620 v3 CPU and 500GB of memory running Ubuntu 16.04 LTS.

Static Analysis. We adopt two state-of-the-art LLVM-based static analysis tools PINPOINT [83] and SVF [84] in the evaluation. They perform sparse value-flow analysis to check the source-sink properties of the program. We use them to detect five common CWE bugs: null pointer deference (NPD), use-after-free (UAF), double-free, file descriptor leak, and memory leak. In our evaluation, all static analysis timeout is set to 24 hours with a memory limit of 500GB.

Metrics. Since our goal is not to evaluate the ability of static analysis tools, we do not directly measure the precision and recall of analysis results. Instead, we take analysis results on the compiled IR as the ***ground truth*** and compare them with that on the lifted IR. Two reports are considered the same if they share the same source and sink locations. ***True positives (TPs)*** denotes vulnerabilities detected in both the compiled IR and the lifted IR. ***False positives (FPs)*** are bugs that are only discovered in the lifted IR. ***False negative (FN)*** represents defects that are only found in the compiled IR.

Benchmarks. As shown in Table 1, we adopt both standard benchmarks and real-world programs in the evaluation. The standard benchmarks include Juliet Test Suite [87]. In terms of real-world programs, we choose 17 open-source C/C++ projects such as `PHP`. In total, we use 18 projects in the evaluation. We use WLLVM [20] (a wrapper for `Clang`) to produce LLVM IRs required by static analysis tools. We build all binaries with two mainstream system compilers `Clang` and `GCC` using project default configurations. Among the adopted 18 projects, 14 of them by default are built with debug information, the remaining 4 programs provide options to enable debug build. Additionally, all binaries by default are built with different levels of compiler optimizations determined by project default configurations, including `-O0`, `-O3`, `-Ofast`, etc. In total, we have 48 binaries in the evaluation.

5.1 Comparison with Existing Lifters

In this section, we compare PLANKTON with five existing binary lifters: RETDEC [11], REOPT [8], MCSEMA [5], REVNG [42], and MCTOLL [9]. Table 1 shows the bug detection results using SVF and PINPOINT on LLVM IRs produced by WLLVM, PLANKTON and compared binary lifters. Since PINPOINT requires the LLVM IR to be 3.6.2 version, which is not supported by the compared lifters, we only adopt SVF in the baseline comparison. Only MCSEMA and PLANKTON successfully lift all binaries (MCSEMA leverages IDA Pro as its front-end). RETDEC, REOPT, and REVNG fail to process 1, 3, and 20 binaries respectively. MCTOLL fails on all binaries because of multiple internal crashes and difficulties in handling external functions [7, 10].

For PINPOINT, PLANKTON achieves a false positive rate of **20.7%** and a false negative rate of **20.9%**. For SVF, the false positive and negative rates achieved by PLANKTON are **13.7%** and **22.1%**. Compared with RETDEC, MCSEMA, REOPT, and REVNG, PLANKTON reduces the false positive/negative rates of SVF by **48.6%/38.5%**, **86.3%/77.9%**, **86.3%/77.9%**, and **86.3%/77.9%** respectively.

RETDEC [11] directly applies debug information without further refinement. Besides, it does not reconstruct complex aggregate types from the debug information. Therefore, SVF can only detect bugs that involve simple data-flows in IRs produced by RETDEC. The above limitations explain its poor performance compared with PLANKTON. REOPT does not output any meaningful static analysis results. The main reason is that REOPT only uses the function boundary in the debug information to assist binary lifting and overlooks others, making the results quite inaccurate. Both MCSEMA

Table 1. Bug detection results for IR produced by WLLVM and lifted from `Clang` compiled binaries by different binary lifters, all results are organized binary-wise. ▼ means the static analysis runs out of memory. ▲ means lifting failure. Only SVF is used for compared lifters since they cannot produce the 3.6.2 version LLVM IR PINPOINT requires. Only the client sides of `MySQL` and `MariaDB` are invoked since both SVF and PINPOINT timeout on their server sides.

Program	WLLVM		PLANKTON						RETDEC			MCSEMA			REOPT			REVNG			MCTOLL
	PINPOINT	SVF	PINPOINT			SVF			SVF			SVF			SVF			SVF			SVF
	#Report	#Report	#FP	#FN	#TP	#FP	#FN	#TP	#FP	#FN	#TP	#FP	#FN	#TP	#FP	#FN	#TP	#FP	#FN	#TP	-
cwe401-1	321	764	83	8	313	55	117	647	192	131	633	0	764	0	0	764	0				
cwe401-2	387	0	77	32	355	0	0	0	0	0	0	0	0	0	0	0	0				
cwe401-3	193	450	53	6	187	36	78	372	120	84	366	0	450	0	0	450	0				
cwe415-1	446	361	28	120	326	15	191	170	126	132	229	0	361	0	0	361	0				
cwe415-2	302	0	32	84	218	0	0	0	0	0	0	0	0	0	0	0	0	▲			
cwe416	884	236	60	1	883	0	0	236	49	35	201	0	236	0	0	236	0				
cwe476	280	51	7	150	130	0	0	51	2	4	47	0	51	0	0	51	0				
bash	3	31	0	2	1	21	8	23	135	11	20	0	31	0	0	31	0				
darknet	21	666	16	6	15	66	57	609	128	163	503	0	666	0	0	666	0	5	666	0	
ffmpeg	196	▼	204	55	141		▼			▼			▼								
git	68	612	24	21	47	136	82	530	879	338	274	0	612	0		▲					
libcrypto.so	114	647	19	42	72	186	25	622	1009	418	229	0	647	0	0	647	0		▲		
libicuuc.so	97	257	63	34	63	35	72	185	179	222	35	0	257	0	0	257	0				▲
libuv.so	0	7	2	0	0	0	0	7	1	6	1	0	7	0	0	7	0	5	7	0	
mariadb	39	86	2	7	32	13	11	75	105	66	20	0	86	0	0	86	0				
mysql	64	111	4	39	25	17	15	96	0	111	0	0	111	0	0	111	0				
php	189	2160	101	66	123	156	781	1379		▲		0	2160	0		▲			▲		
python	31	719	18	12	19	125	145	574	171	717	2	0	719	0	0	719	0				
redis-server	871	911	93	238	633	102	331	580	71	881	30	0	911	0	0	911	0				
ss-local	1	22	10	0	1	2	6	16	16	16	6	0	22	0	0	22	0	5	22	0	
tmux	49	292	15	2	47	26	48	244	153	256	36	0	292	0	0	292	0	5	292	0	
vim	103	536	8	40	63	118	76	460	1335	301	235		▼		0	536	0				
wget	42	53	16	4	38	29	5	48	66	42	11	0	53	0	0	53	0		▲		
wrk	56	516	49	28	28	35	53	463	33	511	5		▼		0	516	0				
Total	**4757**	**9488**	**984**	**997**	**3760**	**1173**	**2101**	**7387**	**4770**	**4445**	**2883**	**0**	**8436**	**0**	**0**	**6716**	**0**	**20**	**987**	**0**	**0**
Rate	-		**.207**	**.209**	**.790**	**.137**	**.221**	**.779**	**.623**	**.607**	**.393**	**1.0**	**1.0**	**.000**	**1.0**	**1.0**	**.000**	**1.0**	**1.0**	**.000**	-

and REVNG produce emulation-style LLVM IRs, which lack too much high-level information and are incompatible with source-level static analysis tools like SVF [63]. Therefore, they do not produce any true positives.

5.1.1 False Cases Analysis. We manually inspect both false positives and negatives in the bug reports and summarize several common reasons. The first one, which is also the prominent problem, is that although static analysis tools are designed to be "deterministic", which means they are expected to produce the same results for the same LLVM IR, they still cannot guarantee consistent outputs for two semantically equivalent but syntactically different IRs [77] (e.g., differently optimized IRs) because of the unsoundness of actual implementations. Our experiments in § 5.4.2 also support this claim. Second, optimizations adopted by PLANKTON could merge or delete call sites, causing different source-sink locations. For example, we found that all false negatives in `CWE415` are caused by optimized away memory allocations and deallocations. Third, despite effort spent implementing PLANKTON, some reverse engineering problems still exist that are not appropriately handled. For example, PLANKTON currently cannot accurately parse virtual tables in C++ classes, causing false cases in C++ code.

The public availability of SVF allows us to dig deeper into the root cause. We found most false cases are caused by an assumption made by SVF: the IR is unoptimized, which does not hold in IR produced by PLANKTON. Optimizations could transform typed pointer arithmetic (**getelementptr**) to arithmetic with void pointer and pointer cast (**bitcast**), which means **getelementptr** does not preserve structure or array information anymore. Although such transformations are valid for code generation, they are currently not handled by SVF's analysis rules, causing different pointer analysis and bug detection results. For example, SVF fails to resolve dozens of memory allocation function pointers in optimized code produced by PLANKTON, which accounts for almost all false negatives. Another major problem is the handling of path conditions. SVF collects path conditions during the source-sink analysis and queries a solver for satisfiability. However, we found that different path conditions collected from syntactically different IRs could cause the solver to produce contrary satisfiability results.

5.2 Generalization to Other Compilers

We also compare the results of checking IRs lifted from `GCC` compiled binaries and compiled IRs [2]. For PINPOINT, PLANKTON achieves a false positive rate of **23.8%** and a false negative rate of **25.7%**. For SVF, the false positive and negative rates achieved by PLANKTON are **24.9%** and **25.8%**. The results are a little bit worse than that of `Clang` compiled

[2]Due to page limitation, detailed data can be found in the supplement material

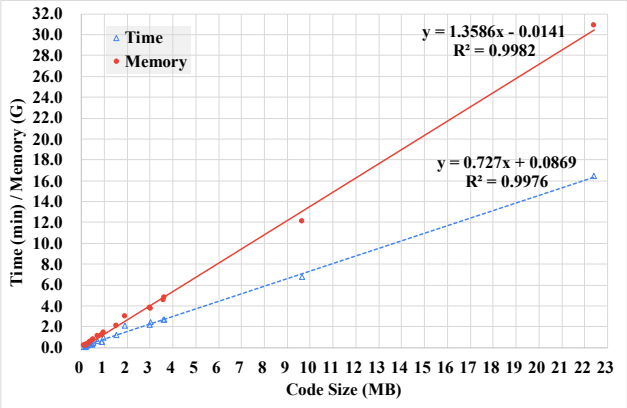


Figure 8. Linear fit lines on the runtime data. The x-axis stands for the size of the code (MB). The y-axis stands for the time cost (minute) or the memory cost (GB).

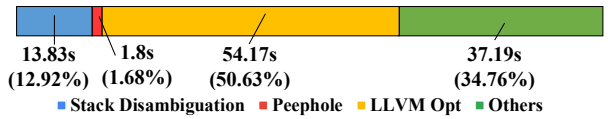


Figure 9. Average execution time breakdown of PLANKTON. Both the data in seconds and the percentage are shown.

binaries. We found that the main reason for the worse performance is inherent differences between compilers, such as inline decisions and optimizations, which could amplify the effect of the unsound static analysis implementations (as shown in § 5.1.1). For example, we found that the GCC compiled `python` does not merge return instructions like `Clang`; by adding an extra transformation that unifies returns, we eliminated 216 false positives. Most false negatives in `redis` are caused by different inline functions. By disabling inlining during compilation, its false negative rate drops from 75.4% to 15%. Therefore, we use `Clang` in the baseline comparison to eliminate the influence of compiler differences and better demonstrate the accuracy of our method. The results also indicate that PINPOINT is less influenced by syntactic differences compared with SVF, with only a 3.9% false rate increase on average compared with 7.4%.

5.3 Scalability

We adopt the curve fitting approach [81] to study the observed time- and memory-complexity of PLANKTON. Figure 8 shows the fitting curves and their coefficients of determination R^2. $R^2 \in [0, 1]$ is a statistical measure of how close the data are to the fitting curve. The closer R^2 is to 1, the better the fitting curve is. It shows that PLANKTON's time and memory cost grows almost linearly in practice ($R^2 > 0.99$), thus scaling up quite gracefully. For example, the code size of `FFmpeg` is 6.1× bigger than that of `git`. Therefore,

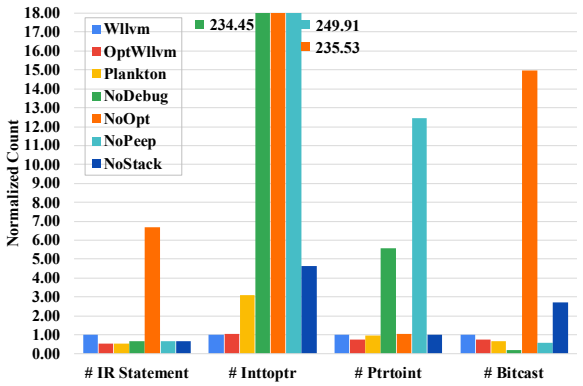


Figure 10. Comparison of instruction count and number of type conversions. All results are normalized against WLLVM. Three columns of data that are too large to show are marked with their actual number.

PLANKTON takes 6.1× more time and consumes 6.4× more memory. For projects with a code size of less than 2MB, the lifting time is less than 2 minutes, and memory consumption is less than 3GB. For the biggest project `FFmpeg`, PLANKTON spends 16.5 minutes and consumes 30.9GB of memory. On average, PLANKTON spends 103 seconds on one binary with 3GB memory consumed, showing good scalability.

Figure 9 presents the time cost breakdown of PLANKTON. The majority of its time is spent on LLVM native optimizations (50.63%). PLANKTON applies all `O3`-level optimizations supported by LLVM to the IR in serial, but this process can be easily parallelized since they are only performed intra-procedurally. Both stack disambiguation and our custom peephole transformations are very efficient, counting for only 14.58% of all the lifting time. Other lifting stages, such as control-flow recovery and function prototype recovery, account for 34.8% of the time, which on average takes 36.03s.

We also compare the memory consumption of PLANKTON with that of RETDEC. We omit the comparison with other lifters because they either fail on too many binaries (e.g., MCTOLL) or depend on results of other tools (e.g., MCSEMA uses IDA Pro during its lifting process). For projects that can be handled by RETDEC, PLANKTON reduces about 11% memory consumption, showing that the two new algorithms do not cause additional memory overhead.

5.4 Ablation Study

In this section, we conduct ablation studies to understand the importance of key components in PLANKTON. Specifically, we consider the following four ablations:

- NODEBUG performs binary lifting without using the debug information.
- NOSTACK does not do stack disambiguation.
- NOOPT does not perform LLVM native optimizations during the type enforcement.

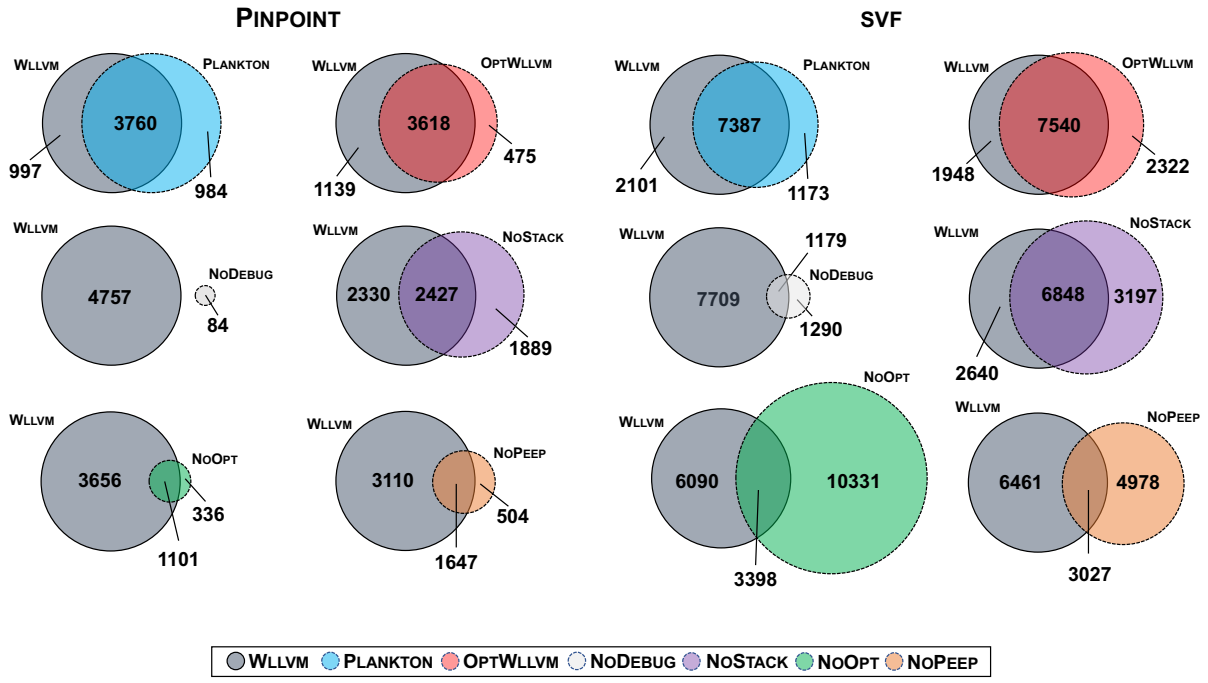


Figure 11. Venn diagrams of the number of bugs detected by PINPOINT and SVF on OPTWLLVM, PLANKTON and its ablations. All results are compared with the default compiled IR (WLLVM).

- NOPEEP does not use peephole transformation rules during the type enforcement.

We also include IRs obtained by re-optimizing IRs produced by WLLVM using the LLVM offline re-optimizer [58], denoted as OPTWLLVM.

5.4.1 IR Quality. We first measure the quality of lifted IRs by two metrics defined by existing work [80]: the size of the IR code and the number of type conversions. Figure 10 presents the results on all benchmarks that are normalized against WLLVM.

NOOPT produces the largest IR code without further code optimization (on average 10.5× bigger than others). All ablations and OPTWLLVM generate smaller IRs compared with WLLVM, indicating the effectiveness of code optimizations in reducing code size. Smaller code could also accelerate static analysis. According to our evaluation, analyzing PLANKTON' IR runs 1.3× faster than analyzing WLLVM's IR.

PLANKTON produces fewer type conversions between integers and pointers (**ptrtoint** and **inttoptr**) than all other ablations. On average, PLANKTON reduces 90.9%, 55.1%, 95.6%, and 19.2% conversions compared with NODEBUG, NOOPT, NOPEEP, and NOSTACK respectively. Without debug information, data-flows and type information cannot be soundly recovered, severely restricting the type correctness level. The results of NOPEEP and NOOPT illustrate the importance of combining both LLVM native optimizations and custom transformation rules, bringing IRs with higher quality than either. We also found that compared with only applying promotion rules, combining promotion and demotion produces 31.4% fewer type conversions in the lifted IR. On average, PLANKTON produces 75.6% fewer **bitcast** instructions than NOSTACK. Such results indicate the effectiveness of the stack disambiguation algorithm in avoiding overlapping stack slot usage, which will otherwise result in heavy use of casting between different variables. PLANKTON produces more casting instructions between pointers (**bitcast**) than NODEBUG and NOPEEP. It is because both of them heavily use **ptrtoint** and **inttoptr** to implement pointer arithmetic, preventing them from casting values to correct types. They even have fewer **bitcast** instructions than WLLVM and OPTWLLVM. On average, only 81% of stack references are mapped to source-level variables by the debug information. However, this percentage can be as low as 35% in some projects (e.g., `Python`) due to compiler optimizations, and the stack disambiguation algorithm can compensate for the remaining ones.

The compiler-generated IR is also not free of type conversions, WLLVM produces 29,398 **ptrtoint** instructions and 3,951 **inttoptr** instructions in all IRs. PLANKTON has fewer **ptrtoint** and **bitcast** instructions compared with WLLVM, but has more **inttoptr** instructions, it is because some assembly instructions and external library functions are not currently handled by PLANKTON, making it unable to transform the IR properly. For example, `tmux` project depends on `libevent` APIs, but debug information of dynamically linked libraries

is not present in the compiled binary. Similar issues also exist in `php`, which links `libxml` dynamically. This problem could be easily fixed by analyzing debug information of external libraries during binary lifting, and we leave it as future work.

5.4.2 Static Analysis Precision. Figure 11 shows the bug detection results of Plankton, OptWllvm, and all ablations. Plankton outperforms all ablations with more true positives (on average 208% more), lower false positive rates (on average 34.6% lower), and lower false negative rates (on average 42.8% lower). NoDebug has the worst performance compared with all the other ablations. Without debug information, NoDebug cannot accurately recover control- and data-flows. Moreover, the incompleteness of the debug information is also not appropriately handled, making it hard for static analyzers to find buggy paths. Without a higher level of type correctness, both NoOpt and NoPeep hinder static analysis capabilities. Therefore, even with high-quality control- and data-flow recovery, NoOpt and NoPeep still perform worse than Plankton.

NoStack performs better than other ablations but still worse than Plankton since variable information is still not fully recovered. Specifically, there are 3,983,477 stack references in all binaries, and 20,9172 of them benefit from the stack disambiguation algorithm. Only 81% of stack references are mapped to source-level variables by the debug information on average. However, this percentage can be as low as 35% in some projects (e.g., `Python`) due to compilation.

We also measure the influence of the may scope inference on the overall results by disabling it in the stack disambiguation algorithm. Without the may scope, static analysis results on SVF generates 11% more false positives, 1.2% more false negatives, and 10.9% fewer true positives, indicating that it has more positive effects on the lifting than negative ones. Similarly, for Pinpoint, disabling the may scope results in 3% and 1% more false positives and negatives, respectively, and 3.1% fewer true positives. The influence is subtle because only a small portion of stack slots needs may scope information, accounting for only 9.3% of all mapped ones.

Both wllvm and OptWllvm use IRs compiled from the source code to check for bugs. The only difference is that OptWllvm additionally re-optimizes all IRs using `O3`-level optimizations. Plankton and OptWllvm produce similar number of true positives (11147 vs. 11158). The overall false positive rate of OptWllvm is even slightly higher than that of Plankton (20% vs. 16.2%), while they have comparable false negative rates (21.6% vs. 21.7%). The above result indicates that static analyzers could still produce diverse results on semantically equivalent but syntactically different IRs. Therefore, the difference between Plankton and wllvm is reasonable and inevitable, showing that Plankton is effective enough for real-world usage.

6 Discussion

Cross-compilation. Plankton also supports other architectures (e.g., `Arm`). In the experiments, we only evaluate with `X86_64` due to its popularity, and all baselines support it.

Producing other IR forms. Currently, Plankton only produces different versions of LLVM IRs. This is because LLVM IR is more suitable for static analysis, and many static analysis tools are already built up on it. Although Plankton cannot directly produce other forms of IRs such as Gimple [38], the underlying algorithms are general so that one can write a Gimple lifter based on them.

Other downstream applications. Since Plankton is able to produce close-to-source LLVM IRs, it can support other applications that are based on LLVM apart from static analysis, including binary composition analysis [64], post-link optimization [72], and code embeddings [88]. Some of them, such as post-link optimization, require the LLVM IR to be recompilable. Although the LLVM IR is recompilable by design, the current Plankton can not guarantee the 100% success of recompilation because not all assembly instructions have their LLVM IR counterparts (some instructions are system specific) and not all library functions are correctly modeled (§ 5.4.1). Therefore, extra efforts are needed for Plankton to support other applications. We leave it as part of our future work.

7 Related Work

Binary Lifting. Many lifters have been proposed to produce LLVM IRs from binaries. SecondWrite [26, 27, 47] uses VSA-based approaches to recover variables and data types from stripped binaries. They do not consider possible variable overlaps and only infer types for top-level variables. BinRec [25] lifts binaries based on dynamic disassembly. Lasagne [80] statically translates `X86_64` binaries to LLVM IR and then compiles it to `Arm` while enforcing the x86 memory ordering model. revng [41, 42] relies on QEMU to perform lifting. LISC [51] automatically learns translation rules from assembly to IR. Inception [40] merges LLVM IRs produced from binaries and source code. mctoll [9] and RetDec [11] also adopt code transformations to refine the produced LLVM IR. However, the peephole transformations adopted by mctoll only consider single operations against integer-typed pointees and LLVM optimizations is only used for reducing the code size. Also, all transformations are only run once without guidance, severely limiting their effectiveness.

Metadata-assisted Binary Analysis. Many existing works have adopted metadata emitted by the compiler to facilitate analysis tasks that require high-quality binary translation results. Egalito [93] and RetroWrite [44] use additional metadata embedded in position-independent code (PIC) to perform accurate binary rewriting. PEBIL [59] and Vulcan [46] require symbol information to perform static binary instrumentation. Zeng et al. [95], Shuffler [92], MCFI [70], Selfrando [39] and, CCR [57] all make use of compiler-generated

information to enforce control-flow intergrity (CFI). HPC-Toolkit [86] proposed to use the debug information to understand the performance of fully optimized modular code. Park et al. [75] leverage JNI-interoperation-related types to improve binary decompilation and static analysis. However, they only handle and propagate function signature types. Unlike them, in PLANKTON, we are the first to use the debug information to facilitate static code analysis.

8 Conclusion

In this paper, we propose a new binary lifter PLANKTON together with two new algorithms that can fill the gap between the low- and high-level code to support full-fledged static analysis. Experimental results show that PLANKTON has sufficient precision and scalability for real-world bug detection.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and opinions for improving this work. This work is supported by the ITS/440/18FP grant from the Hong Kong Innovation and Technology Commission and research grants from Huawei, Microsoft, and TCL. Heqing Huang is the corresponding author.

References

[1] Ida pro. https://www.hex-rays.com/ida-pro/, 2003.
[2] Wllvm fails. https://github.com/SRI-CSL/gllvm/issues/47, 2015.
[3] Coverity scan. https://scan.coverity.com/projects/, 2018.
[4] Coverity scan fails. https://stackoverflow.com/questions/50434236/coverity-scan-fails-to-build-stdlib-h-with-gnu-source-defined/, 2018.
[5] Mcsema. https://github.com/lifting-bits/mcsema/, 2018.
[6] Coverity scan fails. https://github.com/civetweb/civetweb/issues/769, 2019.
[7] mctoll issue. https://github.com/microsoft/llvm-mctoll/issues/55, 2019.
[8] Galois inc. reopt. https://github.com/GaloisInc/reopt/, 2020.
[9] Llvm-mctoll. https://github.com/Microsoft/llvm-mctoll/, 2020.
[10] mctoll issue. https://github.com/microsoft/llvm-mctoll/issues/72, 2020.
[11] Retdec. https://github.com/avast/retdec/, 2020.
[12] Wllvm fails. https://github.com/SRI-CSL/gllvm/issues/39, 2020.
[13] Cppcheck. https://cppcheck.sourceforge.io/, 2021.
[14] Debugging a crashed application. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/developer_guide/debugging-crashed-application/, 2021.
[15] Ghidra. https://github.com/NationalSecurityAgency/ghidra/, 2021.
[16] Infer. https://github.com/facebook/infer/, 2021.
[17] Intel® vtune™ profiler user guide. https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/set-up-analysis-target/linux-targets/debug-info-for-linux-binaries.html, 2021.
[18] Linux debug symbol packages. https://wiki.ubuntu.com/DebugSymbolPackages/, 2021.
[19] Llvm lifetime intrinsic. https://llvm.org/docs/LangRef.html#llvm-lifetime-start-intrinsic, 2021.
[20] Wllvm: whole program llvm. https://github.com/travitch/whole-program-llvm/, 2021.
[21] Debugging tools for windows. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/, 2022.
[22] Windows debug symbols. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/public-and-private-symbols/, 2022.
[23] Build systems for c++. https://hackingcpp.com/cpp/tools/build_systems.html, 2023.
[24] C++ compilers. https://en.wikipedia.org/wiki/List_of_compilers#C++_compilers, 2023.
[25] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, et al. Binrec: dynamic binary lifting and recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.
[26] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 295–308, 2013.
[27] Kapil Anand, Matthew Smithson, Aparna Kotha, Khaled Elwazeer, and Rajeev Barua. Decompilation to compiler high ir in a binary rewriter. *University of Maryland, Tech. Rep*, 2010.
[28] Cristian Assaiante, Daniele Cono D'Elia, Giuseppe Antonio Di Luna, and Leonardo Querzoni. Where did my variable go? poking holes in incomplete debug information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 935–947, 2023.
[29] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*, pages 5–23. Springer, 2004.
[30] Gogul Balakrishnan and Thomas Reps. Divine: Discovering variables in executables. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 1–28. Springer, 2007.
[31] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.
[32] Zoran Budimlic, Keith D Cooper, Timothy J Harvey, Ken Kennedy, Timothy S Oberg, and Steven W Reeves. Fast copy coalescing and live-range identification. *ACM SIGPLAN Notices*, 37(5):25–32, 2002.
[33] Yuandao Cai, Yibo Jin, and Charles Zhang. Unleashing the power of type-based call graph construction by using regional pointer information. In *33nd USENIX Security Symposium (USENIX Security 24)*, 2024.
[34] Yuandao Cai, Peisen Yao, Chengfeng Ye, and Charles Zhang. Place your locks well: understanding and detecting lock misuse bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3727–3744, 2023.
[35] Yuandao Cai, Peisen Yao, and Charles Zhang. Canary: practical static detection of inter-thread value-flow bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1126–1140, 2021.
[36] Yuandao Cai, Chengfeng Ye, Qingkai Shi, and Charles Zhang. Peahen: Fast and precise static deadlock detection via context reduction. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 784–796, 2022.
[37] Yuandao Cai and Charles Zhang. A cocktail approach to practical call graph construction. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):1001–1033, 2023.
[38] Sean Callanan, Daniel J Dean, and Erez Zadok. Extending gcc with modular gimple optimizations. In *Proceedings of the 2007 GCC Developers' Summit*, pages 31–37. Citeseer, 2007.
[39] Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi. Selfrando: Securing the tor browser against de-anonymization exploits. *Proc. Priv. Enhancing Technol.*, 2016(4):454–469, 2016.

[40] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception:{System-Wide} security testing of {Real-World} embedded systems software. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 309–326, 2018.

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

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

[43] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. 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*, pages 1034–1045, 2021.

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

[45] Michael J Eager et al. Introduction to the dwarf debugging format. *Group*, 2007.

[46] Andrew Edwards, Hoi Vo, and Amitabh Srivastava. Vulcan binary transformation in a distributed environment. 2001.

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

[48] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 463–474, 2020.

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

[50] Samuel Z Guyer, Kathryn S McKinley, and Daniel Frampton. Free-me: a static analysis for automatic individual object reclamation. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 364–375, 2006.

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

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

[53] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 9–14, 2007.

[54] Tiago Trevisan Jost. *Compilation and optimizations for variable precision floating-Point arithmetic: from language and libraries to code generation.* PhD thesis, Université Grenoble Alpes [2020-....], 2021.

[55] Vini Kanvar and Uday P Khedker. Heap abstractions for static analysis. *ACM Computing Surveys (CSUR)*, 49(2):1–47, 2016.

[56] Brian W Kernighan and Dennis M Ritchie. The c programming language. 2002.

[57] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 461–477. IEEE, 2018.

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

[59] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 175–183. IEEE, 2010.

[60] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P Lopes. Reconciling high-level optimizations and low-level code in llvm. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[61] Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41:1–31, 2008.

[62] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. Debug information validation for optimized code. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1052–1065, 2020.

[63] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. 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*, pages 453–472, 2022.

[64] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering*, 43(12):1157–1177, 2017.

[65] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. 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*, 2022.

[66] Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd international conference on software engineering*, pages 141–150, 2011.

[67] Julia Menapace, Jim Kingdon, and David MacKenzie. The" stabs" debug format. Technical report, Technical report, Cygnus support, 1992.

[68] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):527–568, 1999.

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

[70] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 577–587, 2014.

[71] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 475–484, 2014.

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

[73] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 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)*, pages 833–851. IEEE, 2021.

[74] Chengbin Pang, Tiantai Zhang, Ruotong Yu, Bing Mao, and Jun Xu. Ground truth for binary disassembly is not easy. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2479–2495, 2022.

[75] Jihee Park, Sungho Lee, Jaemin Hong, and Sukyoung Ryu. Static analysis of jni programs via binary decompilation. *IEEE Transactions on Software Engineering*, 2023.

[76] Quoc-Sang Phan, Kim-Hao Nguyen, and ThanhVu Nguyen. The challenges of shift left static analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 340–342. IEEE, 2023.

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

[78] Zvonimir Rakamarić and Alan J Hu. 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*, pages 290–304. Springer, 2009.

[79] Valentin Robert and Xavier Leroy. A formally-verified alias analysis. In *International Conference on Certified Programs and Proofs*, pages 11–26. Springer, 2012.

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

[81] LA Sandra. Phb practical handbook of curve fitting. *ed: CRC Press: Boca Raton, FL, USA*, 1994.

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

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

[84] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[85] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.

[86] Nathan R Tallent, John M Mellor-Crummey, and Michael W Fagan. Binary analysis for measurement and attribution of program performance. *ACM Sigplan Notices*, 44(6):441–452, 2009.

[87] Juliet Test. The juliet 1.1 c/c++ and java test suite. 2012.

[88] S VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and YN Srikant. Ir2vec: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–27, 2020.

[89] Freek Verbeek, Joshua Bockenek, Zhoulai Fu, and Binoy Ravindran. Formally verified lifting of c-compiled x86-64 binaries. In *2022 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2022.

[90] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. Sound c code decompilation for a subset of x86-64 binaries. In *International Conference on Software Engineering and Formal Methods*, pages 247–264. Springer, 2020.

[91] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 522–536. Springer, 2011.

[92] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code {Re-Randomization}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 367–382, 2016.

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

[94] Yichen Xie and Alex Aiken. Context-and path-sensitive memory leak detection. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 115–125, 2005.

[95] Dongrui Zeng and Gang Tan. From debugging-information based binary-level type inference to cfg generation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 366–376, 2018.