What You Trace is What You Get: Dynamic Stack-Layout Recovery for Binary Recompilation

Fabian Parzefall
University of California, Irvine
Irvine, California, USA
fparzefa@uci.edu

Chinmay Deshpande
University of California, Irvine
Irvine, California, USA
cddespha@uci.edu

Felicitas Hetzelt
University of California, Irvine
Irvine, California, USA
fhetzelt@uci.edu

Michael Franz
University of California, Irvine
Irvine, California, USA
franz@uci.edu

Abstract

Users of proprietary and/or legacy programs without vendor support are denied the significant advances in compiler technologies of the past decades. Adapting these technologies to operate directly on binaries without source code is often infeasible. Binary recompilers attempt to bridge this gap by “lifting” binary executables to compiler-level intermediate representations (IR) and “lowering” them back down to executable form, enabling application of the full range of analyses and transformations available in modern compiler infrastructures. Past approaches could not recover local variables in lifted programs with sufficient precision, which is a necessary prerequisite for many compiler-related applications, including performance optimization. They have relied on heuristics failing on certain input programs, or on conservative over-approximations yielding imprecise results.

In this paper, we present a novel approach, WYTIWYG, to recover function-local variables within lifted binaries. Our approach is fully automated and preserves functionality for user-provided inputs. This is accomplished by decomposing the recovery of local variables into a series of instrumentation-based dynamic binary analyses. We conduct an extensive set of careful evaluations on the SPECint 2006 benchmark suite, including direct comparisons with two previously published state-of-the-art binary recompilers. Our approach recompiles fully optimized commercial off-the-shelf binaries compiled with the latest compilers. Using performance of recompiled binaries as an indicator of IR-quality, our approach significantly outperforms similar recompilers by 1.18x on average. Furthermore, WYTIWYG accelerates legacy binaries optimized by older compilers by an astounding 1.22x.

CCS Concepts: • **Software and its engineering** → **Software reverse engineering**; **Dynamic analysis**.

Keywords: binary analysis, binary lifting, binary rewriting, recompilation, stack analysis, variable recovery

ACM Reference Format:
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 *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, 14 pages. https://doi.org/10.1145/3620665.3640371

1 Introduction

Over the past decades, open compiler infrastructures have seen enormous investments by both academic researchers and industry users to advance the analysis, optimization and safety of software. Unfortunately, these advances are often denied to users of legacy binaries. Binaries of programs that can no longer be recompiled from their source code are effectively “stuck in time”. This happens when a vendor ceases support of the software, toolchains are unavailable, or the program's source code has been lost. Naturally, not being able to recompile software makes maintenance of legacy binaries very challenging. For instance, users of legacy binaries cannot reoptimize them to utilize features of recent CPUs, easily fix known bugs and vulnerabilities, or deploy sanitizers and mitigations that are readily available in existing compilers. At the same time, replacing legacy software can be very expensive and is often infeasible.

This issue is addressed by a wide body of research in end-to-end static binary rewriting [22]. Many of the existing approaches are quite effective and capable of applying all kinds of program transformations to commercial off-the-shelf (COTS) binaries. However, most binary rewriting frameworks have a narrow scope and support only a small subset

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

ACM ISBN 979-8-4007-0385-0/24/04
https://doi.org/10.1145/3620665.3640371

of program transformations compared to compilers. State-of-the-art rewriters, such as Egalito [23], support alteration of the program's control-flow graph (CFG) and modification of linear instruction sequences, but provide no help in manipulating accesses to variables and their layout in memory.

Binary recompilers attempt to bridge the gap between rewriters and compilers by "lifting" binaries to compiler-level intermediate representations (IRs). Unlike most IRs used by rewriters, compiler-level IRs like LLVM IR [13] encode source-level program structures, such as local and global variables, and types of variables and functions. However, like rewriters, state-of-the-art recompilers usually omit the recovery of these structures. This is unsurprising, since recovery of variables from binaries is inherently undecidable [10] and a problem of ongoing research [14, 15, 21, 25]. Hence, the programs lifted by recompilers like McSema [7], Rev.ng [6] and BinRec [1] are oblivious to global and local variables, function arguments, saved registers, and spills. Without this information, the efficacy of many program analysis and transformation techniques is severely diminished. Any transformations that affect the program's memory-layout (e.g., AddressSanitizer [19]) cannot be applied to local or global variables. Additionally, program analysis techniques that operate around mappings of variables to sets of their possible values cannot scale without this information.

To overcome these shortcomings, binary lifters augment direct references to stack memory and global data with symbols that denote distinct variables. This process is also called *symbolization* and is supported to some extent by several binary lifters via static program analysis. However, RetDec [12] and mctoll [24] cannot successfully recompile any of the SPECint 2006 benchmarks [16]. The authors of McSema [7] have conveyed to us that recompilation itself is not reliable and requires a skilled operator to fix issues manually. SecondWrite [2] implements a more comprehensive analysis compared to the other tools and offers better compatibility. Nevertheless, we found that, because of the conservative nature of their analysis, SecondWrite associates all local variables of functions beyond a certain complexity with a single symbol rather than recovering individual variables.

Lifters using static analysis for variable recovery are forced to choose between preserving program functionality and achieving fine-grained variable recovery. The former prevents lifting of the binary to an IR that details precise data-dependencies, inhibiting further processing of the program. The latter is prone to alter program semantics in unpredictable ways, which introduces subtle bugs that are amplified by subsequent optimizations. Because of these limitations, recompilers fall short on the promise of delivering the advances of compilers and their ecosystem to binaries. This has been confirmed in a study by Liu et al. [16], in which they investigated how programs lifted by various binary-to-LLVM lifters and recompilers perform in various downstream applications.

In this paper, we propose a dynamic approach, WYTIWYG, to identify and symbolize local variables in functions within lifted COTS-binaries. To facilitate this, WYTIWYG employs an instrumentation-based approach that tracks pointers to stack variables throughout the program and observes how the program derives new pointers from existing ones. Unlike static approaches, relying on dynamic observation of real executions allows us to symbolize functions with high precision while preserving all semantics that are exhibited by the traced inputs.

Our prototype is implemented on top of BinRec [1], a binary recompiler using execution traces to lift programs to LLVM IR. We chose BinRec because it is the only lifter that can reliably recompile SPECint 2006 programs. We leverage the existing lifting capability to instrument and rewrite the lifted programs in an iterative process we call *refinement lifting*. Through a series of refinement iterations, WYTIWYG identifies the stack-layout of lifted functions and symbolizes all references to local variables. To our knowledge, WYTIWYG is the first system using dynamic analysis to symbolize lifted programs.

We evaluate our implementation in three ways. First, we verify the ability of WYTIWYG to symbolize programs from the SPECint 2006 benchmark suite while retaining their functionality. Each target benchmark is compiled in multiple configurations with different compilers and optimization levels to ensure a broad range of inputs. Second, we estimate the "quality" of the generated IR. Since there is no agreed-upon benchmark to measure IR-quality, we rely on performance measurements of the recompiled binaries as a proxy indicator for IR-quality. Language frontends strive to produce IR that is best understood by passes that are part of LLVM, and developers within the LLVM ecosystem optimize their downstream applications to process IR emitted by those frontends. Since LLVM's primary purpose is program optimization and efficient code generation, we argue that a decrease in runtime overhead indicates that the refined programs better match the expectations of LLVM and LLVM-based tools. Last, we conduct a comparative analysis of the inferred stack layouts against the ground-truth data provided by LLVM to ascertain the accuracy of our approach.

To summarize, we make the following key contributions:

- We present the first binary recompiler capable of symbolizing local variables in lifted COTS-binaries precisely while preserving the semantics of the input program.
- We utilize a novel technique, refinement lifting, that employs a series of dynamic analysis passes and successive transformations to canonicalize and symbolize binaries iteratively.
- We evaluate our implementation on a set of real-world benchmarks and compare it to existing state-of-the-art approaches. We demonstrate that our approach han-

dles real-world programs reliably while significantly improving IR-quality. Our fine-grained stack partitioning allows LLVM to generate recompiled programs whose performance competes with the fully optimized input binaries (on average 1.06*x* to 1.10*x* runtime for Clang 16 and GCC 12.2 binaries, respectively) and significantly outperform binaries generated by any other binary recompiler. Further, legacy programs generated by older compilers (GCC 4.4) can be effectively reoptimized, speeding them up by 1.22*x* on average. Finally, we show that our technique identifies stack variables with a precision of 94.4% and a recall of 87.6%.

2 Background

2.1 Binary Recompilation

State-of-the-art binary recompilers usually operate in three phases. First, the target program is analyzed for its control-flow graph (CFG). Recovery of the CFG can be performed both statically [6, 7] and dynamically [1]. Then, the CFG is used to lift the program to a compiler-level IR (all state-of-the-art recompilers target LLVM IR). The operator of the recompiler can now apply the desired transformations and optimizations to the lifted program. And finally, the program is recompiled into a new binary.

One key challenge of lifting binaries is the translation of machine instructions to semantically equivalent instructions expressed in the compiler-level IR. Most machine instructions cannot be directly mapped to IR instructions. Modern CPUs comprise numerous instructions that modify multiple registers all at once, operate on processor-internal state, and/or are implicitly affected by status or control registers (e.g., the x87 floating point stack). Compared to that, compiler IRs usually comprise very few instructions that have no side effects and declare all their dependencies explicitly.

Recompilers solve this by generating a new program that effectively *emulates* the semantics of the input binary. The lifter translates each machine instruction into a sequence of IR-instructions that replicate its effects on a virtual CPU embedded in the lifted program. For instance, a 32-bit x86 push instruction is lifted to an instruction-sequence, that (1) subtracts 4 from the emulated esp register, (2) loads the pushed operand, (3) stores the operand to the address in esp, and (4) sets the emulated program counter to the next instruction's address. Although this technique initially increases the size of the program significantly, most generated instructions are redundant and can be eliminated. For example, the virtual program counter is updated within every translated instruction-sequence, but is only used to compute addresses of global variables or determining the targets of indirect control flows.

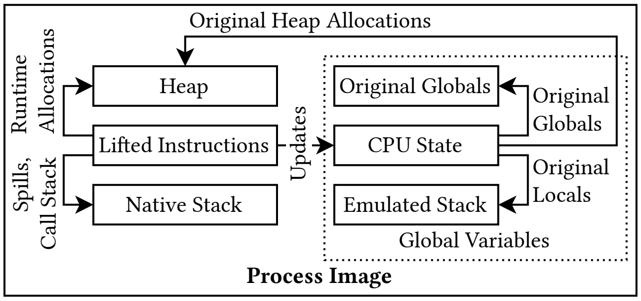


Figure 1. Process image of a recompiled binary.

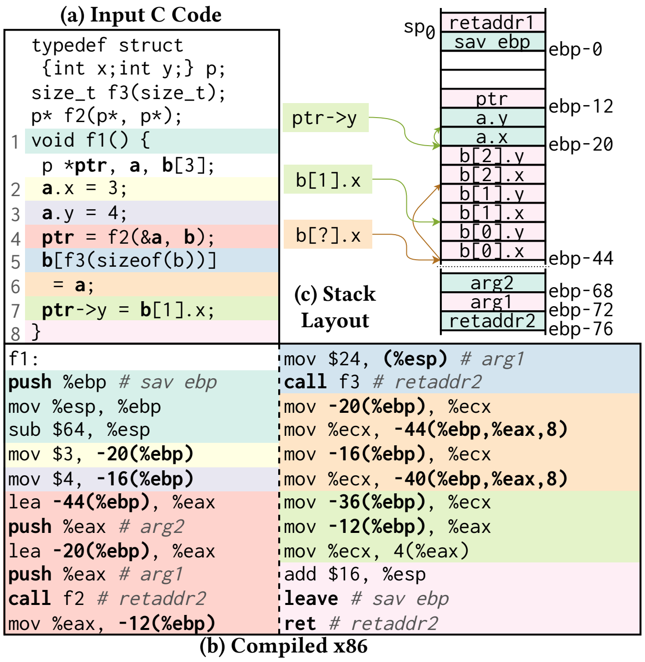
In order for this approach to work, the lifted program needs two stacks: a "native stack" that is used by the "emulator" and holds the call stack of the lifted program, and a pre-allocated byte-array, that is used as "emulated stack" and holds the original program's call stack and local variables (see Figure 1). In this model, the lifted code emulates calls, function prologues and epilogues, spills, and local variables of the original program through the virtual stack pointer.

Maintaining two stacks naturally has some overhead, such as increased register pressure of tracking two stacks and setting up two frames for every function call. More importantly, lifting the stack as an opaque byte array severely limits the recompiler's ability to reason about the program. Consider the assignment and use of ptr in lines 4 and 7 of Figure 2. Since out-of-bound accesses are undefined behaviour, the source-compiler can assume that the access to b in line 5 cannot alias the value of ptr. This information is lost during compilation. Now, in order to infer that the loaded value of ptr in line 7 is identical to the spilled value in line 5, the recompiler has to prove that ebp-44+f3(24)*8 cannot alias with ebp-12. Depending on the complexity of the operations involved in the address computation, this analysis quickly becomes challenging.

Crucially, this prevents generation of precise use-define chains between reloaded values and the sites at which they are spilled. Although the write through ptr->y can be linked with the value returned by f2, the compiler also has to consider indirect writes as potential definitions. Because of this, an alias analysis for example cannot narrow down that access to variables a and b, which diminishes any ability to reason about the program even further. This analysis hazard affects not only pointers, but any complex expression spanning multiple instructions involving values loaded from the stack. We found this to be an issue in all binary recompilers and identified it as the primary cause limiting the efficacy of program analyses and transformations. Liu et al. have confirmed this finding in their study of binary recompilers [16].

2.2 Stack Symbolization

Stack symbolization is the process of labeling direct references to the stack with symbols that denote distinct local variables. Direct references to local variables are only found within the program text of the function owning the frame. They are indirectly encoded as a series of constant offsets rel-



(a) Input C Code

```
  typedef struct
   {int x;int y;} p;
  size_t f3(size_t);
  p* f2(p*, p*);
1 void f1() {
   p *ptr, a, b[3];
2  a.x = 3;
3  a.y = 4;
4  ptr = f2(&a, b);
5  b[f3(sizeof(b))]
6   = a;
7  ptr->y = b[1].x;
8 }
```

(b) Compiled x86

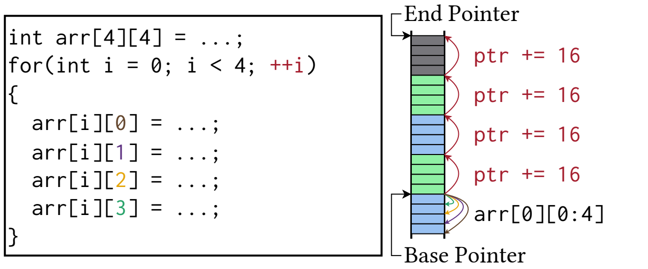
```
f1:
push %ebp # sav ebp
mov %esp, %ebp
sub $64, %esp
mov $3, -20(%ebp)
mov $4, -16(%ebp)
lea -44(%ebp), %eax
push %eax # arg2
lea -20(%ebp), %eax
push %eax # arg1
call f2 # retaddr2
mov %eax, -12(%ebp)
mov $24, (%esp) # arg1
call f3 # retaddr2
mov -20(%ebp), %ecx
mov %ecx, -44(%ebp,%eax,8)
mov -16(%ebp), %ecx
mov %ecx, -40(%ebp,%eax,8)
mov -36(%ebp), %ecx
mov -12(%ebp), %eax
mov %ecx, 4(%eax)
add $16, %esp
leave # sav ebp
ret # retaddr2
```

(c) Stack Layout

Figure 2. An example function and its stack frame. f2 returns one of its arguments. f3 returns a value less than its first argument. Bold values in (a) and (b) refer to the stack. For (c), assume f2 returned &a, and f3 returned 2.

ative to the initial value of the stack pointer sp$_0$ at the start of a function. For example, consider the array-access b[1] in line 7 in Figure 2. The function computes the address to this element using the value of ebp as base. The ebp register itself holds the value sp$_0$-4. Hence, the pointer computed by this instruction can be expressed as sp$_0$-4-36. To symbolize this access, the stack frame has to be partitioned into individual variables. Ideally, an analysis would determine that the frame contains an array $\hat{b}$ at an offset of sp$_0$-4-44 with a size of 24 bytes. Using this information, the aforementioned reference ebp-36 can be labelled with an expression relative to the recovered variable $\hat{b}$+8.

Despite the apparent low complexity of this function, it is remarkably difficult to identify distinct local variables. Accesses to members of composite types, such as in lines 2 and 3, are folded into direct offsets to the frame pointer in optimized binaries and do not reveal their underlying structure. For example, in order to determine the bounds of variable a, an analysis has to establish that the access through ptr to ebp-20 in line 7 can refer to the 8 byte memory area allocated to a. If this condition is met, the offset of 4 and the following write reveal a's total size of 8. At the same time, the indirect access to b in line 5 might access a or any other object in the frame, unless an analysis can provide explicit bounds for the return value of f3. As mentioned in Section 2.1, this is often not possible. If the bounds of the access cannot be determined, conservative static approaches are forced to label all references to local variables of a function with a single symbol.



```
int arr[4][4] = ...;
for(int i = 0; i < 4; ++i)
{
  arr[i][0] = ...;
  arr[i][1] = ...;
  arr[i][2] = ...;
  arr[i][3] = ...;
}
```

Figure 3. Optimizations can transform the loop from incrementing an index to incrementing the base pointer, such that the end pointer is located past the array. The individual elements are accessed with a negative displacement.

Even if the stack frame has been perfectly partitioned into its individual variables, labeling all references in the function with the correct symbol is another challenge. In C and C++, any expression that results in a pointer that is out-of-bounds relative to its underlying array is undefined behavior, even if the pointer is not dereferenced [11]. However, that does not prohibit compilers from generating code that computes pointers lying outside the objects they refer to. For instance, compilers can turn certain index-based iterations over arrays into pointer-based iterations. Combined with other optimizations, the "end"-pointer that is used in the termination condition of such loops points, in rare cases, outside its corresponding array. Figure 3 illustrates what the access pattern of such a case looks like. Consequently, direct references to the stack cannot be automatically associated with variables that are allocated at the same position.

3 Overview

Figure 4 illustrates WYTIWYG's binary recompilation process in two phases. First, we rely on BinRec [1] to recover the target binary's CFG. BinRec uses a binary tracer (S2E [5]) that records all control transfers of the program with a user-provided set of inputs. Based on the CFG, a machine code to LLVM translator lifts the binary to LLVM IR using the instruction emulation approach outlined in Section 2.1. This program can already be recompiled, but it lacks variable information.

Our main contribution comprises the symbolization phase, which is outlined in gray in Figure 4. We split the symbolization process into multiple steps with dedicated dynamic analyses and IR-level transformations. This is similar to how certain passes are used to simplify and canonicalize the IR in regular compiler pipelines. Because this process refines the lifted program's IR and improves its quality in every iteration, we call this approach *Refinement Lifting*.

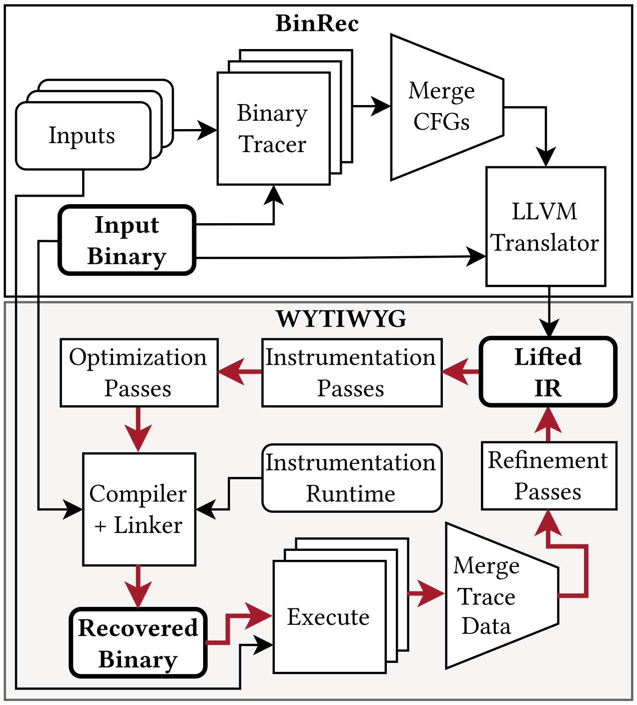


Figure 4. Overview of WYTIWYG. The upper section corresponds to the original BinRec recompiler. The lower section outlines our contribution. The red highlighted transitions correspond to the *Refinement Lifting* process.

In this phase, WYTIWYG iteratively symbolizes local variables in each function using dynamic analyses. Because the lifted program can be modified like any other LLVM program, we can easily instrument the lifted IR and implement all our dynamic analyses in an external library (similar to sanitizer implementations in LLVM). At the end of each iteration, the program is transformed according to the analysis results, such that they are immediately available in the next iteration.

4 Dynamic Stack Symbolization

To symbolize local variables, WYTIWYG employs two refinements. The first identifies all direct stack references and rewrites them into expressions relative to sp_0. The second then determines the maximal offsets of pointers derived from each direct stack reference, uses this information to compute a stack layout for each function, and labels all direct references with symbols referring to variables within the stack layout.

4.1 Stack Reference Identification

To symbolize local variables comprehensively, we first need to identify all values throughout the program that constitute a direct reference to the program's stack memory. As explained in Section 2.2, direct stack references are pointers within a function that are computed as a sequence of constant displacements to sp_0. By folding all uses of sp_0, we can identify all direct stack references, and simplify them by replacing them with expressions of the form sp_0+offset. Consider the push instruction corresponding to arg1 of the call in line 4 in Figure 2. It is initially lifted to this pseudo-IR:

```
@vcpu.esp = @vcpu.esp - 4;
*@vcpu.esp = @vcpu.ebp - 20;
```

After identifying all displacements, these instructions are replaced with the following expression:

$$*(\%sp_0 - \overbrace{4}^{\texttt{push \%ebp}} - \underbrace{64}_{\texttt{sub \$64, \%esp}} - \overbrace{4}^{\texttt{push \%eax \# arg2}} - \underbrace{4}_{\texttt{push \%eax \# arg1}}) = \%sp_0 - 28;$$

However, not all uses of sp_0 can be trivially simplified, since registers holding intermediate stack references are frequently spilled onto the stack in function prologues and epilogues. In our example, f1 saves and restores the ebp register during the first push and the leave instructions. From the recompiler's perspective, it is not apparent that the value of ebp is preserved across the invocation of f1. Instead ebp appears to be assigned an opaque value loaded from memory before returning from the call. If ebp holds an intermediary stack reference (e.g., the frame pointer of the calling function) before the call, none of the pointers derived from it after the call can be folded into an offset of sp_0.

To address this, we determine for each register used in a function, whether it is merely saved on the stack for the duration of the call, or whether it is part of the function's signature. Unfortunately, indirect accesses, as for example the write to b in line 5, could modify any value stored on the stack and therefore complicate determining whether ebp is a saved register (refer to Section 2.2).

Saved registers are often identified through heuristics, that rely on platform ABI conventions to codify, which registers are to be saved to the stack before they can be used in a function, and which registers are used to transfer arguments and return between the caller and the callee (such as the System V ABI [17]). However, compilers (and sometimes developers) can disregard these conventions for functions that are not exported to other translation-units and define their own conventions on a per-function basis. Additionally, if function recovery cannot be performed with perfect accuracy, registers might not be saved and restored at the start and end of the function, and they can be saved multiple times. This can happen when tail-called functions are merged into their caller (refer to Section 5.1). For these reasons, identifying stack references based on heuristics is not reliable.

To avoid these issues, we use a dynamic analysis instead. Upon function entry, we assign each register a symbolic value and track how this value is used throughout the function. We consider a register saved, if the following conditions are met:

- Its symbolic value is only used in load- and store-operations from and to the current function's stack frame. If the symbol is written to any other location or used in any operation, we treat the register as an argument to the function.
- When the function returns, the virtual register contains its initially assigned symbol.

Sometimes, a register used to pass an argument is not explicitly used throughout the called function's entire body, but is "forwarded" to another function. For example, in Figure 2b, the register edx is not used once. Without knowledge of function signatures, f2 could either save edx to the stack, or use it as an argument. If edx is used as an argument within f2, we also need to make it an argument to f1. In a situation like this, we examine a register's usage within the function it is forwarded to in order to determine whether it is saved. We consider each forwarded register as saved, unless we classify it as an argument in any of the functions it is passed to.

Since registers can be forwarded through multiple functions until they are used, we defer evaluating the state of forwarded registers until tracing is complete. During tracing, we only record whenever a register symbol is forwarded to another function. Afterwards, we use this information to generate constraints for each forwarded register. In our example, we would produce the constraint "if edx is used as an argument in f2, then it is also an argument to f1". If that constraint is fulfilled, edx will be explicitly marked as an argument to f1.

Having identified saved registers for all functions in the binary, we preemptively save and restore these registers at all call sites:

```
%tmp_ebp = @vcpu.ebp
call f1   # saves ebp
@vcpu.ebp = %tmp_ebp
```

This transforms the indirect dependency on the value of ebp saved to and restored from the stack within f1 into a direct dependency on the register's value %tmp_ebp from before the call. This IR refinement therefore substantially simplifies the identification of stack references through register spills. After folding all constant offset to $\texttt{sp}_0$, all direct references to objects on the emulated stack are expressed in terms of $\texttt{sp}_0$. These rewritten references serve as "base pointers" to local variables in the next refinement.

4.2 Object Bounds Recovery

Having identified all direct stack references, this refinement's purpose is to determine the layout of each stack frame and assign stack references to the identified variables. WYTIWYG uses a bottom-up approach to divide a stack-frame into distinct variables. At this point, it is unknown which references refer to the same object. Hence, we initially consider each stack reference provided by the previous refinement as a base pointer to a distinct local variable. Then, we use a dynamic analysis to record the relative minimum and maximum offsets of pointers derived from each base pointer. This yields an interval for each base pointer that indicates the underlying object's size. Expressing these intervals as ranges in terms of $\texttt{sp}_0$ yields continuous sections within each stack frame that belong to the same variable.

To generate the stack layout, we merge all ranges that are overlapping with each other and assign their associated base pointers the same symbol. For example, consider the references %ebp-44 and %ebp-36 to variable b in lines 6 and 7 of Figure 2. Initially, we assume that these two pointers belong to different objects. Once the dynamic analysis observes an access to the third element of the array, the former pointer's interval will be recorded as [0;20] (offset of 16 and access size of 4). Since this subsumes the latter pointer's interval [0;4], they belong to the same object.

This also means that if f3 returns 0 in every invocation across all traces, the array will be split into two distinct symbols. Since the generated layouts are the product of actual executions, this approach ensures, for the provided set of inputs, that all base pointers are associated with the correct symbol and that the symbolized variables are sufficiently large without causing unpredictable out-of-bounds accesses (see also Section 7.2).

4.2.1 Tracing Runtime Overview.

To track direct and indirect stack references, we employ a runtime, which is illustrated in Figure 5. We associate every previously identified base pointer with a unique id. For every id, we allocate a StackVar within our runtime, which records the bounds of the corresponding base pointer. We do not track the address of the associated base pointer in its StackVar, because one StackVar can be associated with the same variable in multiple stack-frames of the same function in recursive call-chains. As the program executes and derives new pointers from existing ones, the instrumented binary informs the runtime to update the bounds of individual StackVars.

To track whether an LLVM-value refers to a StackVar during execution, we associate each LLVM-instruction with a PointerInfo. Apart from a pointer to the currently referenced StackVar, this metadata also records the offset from the variable's base pointer. We allocate these PointerInfo objects for each function on the native stack, because a single logical LLVM-value can point to multiple different objects in recursive calls to the same function. Since x86 does not distinguish between pointers and integers, it is not always possible to determine statically whether a value loaded from memory is a pointer. Hence, we track this metadata for every pointer-sized integer. Additionally, we maintain a mapping of memory-addresses to PointerInfo which is updated whenever a pointer to a stack variable is written to or read from memory.

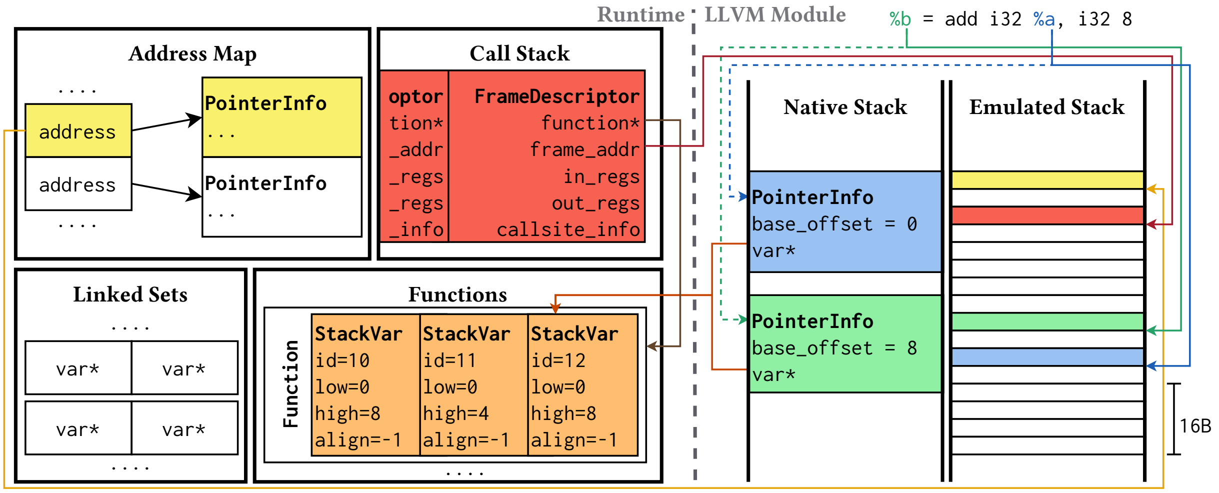


Figure 5. Overview of our tracing runtime.

4.2.2 Core Tracing Operations. In the following, we explain the central runtime operations that we instrument lifted programs with. The arguments to all operations refer to tracked LLVM instructions, which are pairs of their concrete values (i.e., the result of the instruction) and their associated PointerInfo metadata. The operations we implement to track pointers are:

- **derive**(derived, base) indicates that derived is a value derived from base.
- **derive2**(derived, lhs, rhs) is similar to **derive**, but is used when derived refers to a binary operator for which both operands could be pointers to stack variables.
- **link**(a, b) marks that the operands a and b belong to the same stack variable.
- **store**(value, pointer) records that any stack variable reference contained by value is written to the address specified by pointer.
- **load**(value, pointer) indicates that value contains any stack variable reference previously written to the address specified by pointer.
- **copy**(dst, src) assigns dst the pointer info of src.

We use **derive** to instrument pointer-sized add, sub and and instructions that have one constant operand. If base is associated with a StackVar, we initialize the metadata of derived accordingly. Additionally, for and instructions, we capture the alignment factor in the associated StackVar.

If an add or sub instruction does not have a constant operand, we instrument it with **derive2**. If exactly one of its operands points to a stack variable at runtime, we forward the arguments to **derive** with the known pointer operand as base. If both operands of a sub instruction are pointers, the result is the difference between the two pointers. Here, we call **link** instead to record that both pointers belong to the same variable. The same applies to cmp instructions. Linked variables are stored by the runtime as pairs in a hash set.

The **load** and **store** operations are inserted for the LLVM-instructions of the same name. They record any pointers to stack variables that are written to memory by updating the *Address Map*. Additionally, if we load or store from or to a stack variable, we update the upper bound of the StackVar associated with the pointer by the size of the memory access (e.g., a store of a 32-bit integer to a pointer will update its upper bound to 4).

To simplify the program, we turn virtual CPU registers into SSA-values before instrumentation. Hence, we need one last core operation **copy** to support PHI-nodes. This operation is used to copy the PointerInfo of incoming values to the PHI-node's associated PointerInfo. We insert these at the predecessor-blocks for each phi instruction.

4.2.3 False Derives. Deriving a value from a pointer does not always yield a new pointer. Writing to 16-bit or 8-bit x86 sub-registers does not zero out their upper 16 or 24 bits, respectively. This is prevalent in C++ code using booleans. Because there can be arbitrarily complex code between writing to and reading from such a register (including crossing function boundaries), it might not be possible to determine that the upper bytes of a register are stale. This creates a false dependency on the value previously stored in the register. This is problematic if the previous value is a tracked pointer, because the result of a subregister store appears to be a pointer, but might be an entirely unrelated value. The only way to confirm the value's validity as a pointer is to wait until it is dereferenced. For this reason, we do not update the bounds of stack variables in the **derive** operation, but only within the **load** and **store** operations.

4.2.4 Out-of-Bound Pointers. Deferring updates to the bounds of stack variables until a pointer is dereferenced addresses out-of-bound pointers only partially. It correctly handles the case exemplified in Figure 3. However, sometimes, the base pointer itself can be out-of-bounds of the variable it refers to. In that case, we need to defer *intialization* of the bounds until the first access, instead of automatically assuming that the base pointer is part of the object. Hence, we update bounds according to the following conditions:

- The bounds of `StackVars` are initially undefined.
- The first time any pointer associated with a `StackVar` is dereferenced, the lower *and* upper bounds are initialized with that pointer's offset.
- When linking two `StackVars`, their ranges are only merged if both have defined bounds.

4.2.5 Function Calls. Arguments to functions are usually pushed to the stack by the caller. Hence, we not only observe direct accesses to the stack frame of the current function, but also to the frame of the function invoking it. Because functions can have a variable number of arguments, we record such accesses per call-site.

To facilitate this, the runtime keeps track of the call stack and records, for each stack frame, its base pointer (i.e., $\mathtt{sp}_0$) and the call site through which the function was entered. A call site descriptor comprises, similar to a function, a list of stack variables, a `PointerInfo` referring to the stack variable containing the argument list in the original frame, and an `id` that lets us map it to its corresponding LLVM call-instruction. The **`derive`** operation treats all base pointers with an offset greater or equal to the current frame's $\mathtt{sp}_0$ as arguments. These accesses are recorded in the call site descriptor of the currently active frame.

To keep track of the current call-stack in our runtime, we instrument every LLVM call instruction with the following operations:

- **`fnenter`**`(function*, callsite, regs)` creates a new frame descriptor for `function` called at `callsite`.
- **`fnexit`**`(regs)` pops the current stack frame from the runtime call stack.

Both operations additionally have a list of registers as arguments to marshal metadata associated with virtual registers between calls.

4.2.6 Replacing Base Pointers. Tracing yields bounds and alignments for base pointers, argument lists for call sites, and a list of linked base pointers. After coalescing overlapping and linked base pointers, we generate function signatures. First, we merge all call site signatures for a function into a *super signature*. We propagate the super signature back to all call sites to fill in gaps in argument lists. For example, consider a call site to a function with four arguments. Suppose we only traced accesses to the first and third argument. In this case, we would fill in the second argument. We omit the fourth argument, because it could be a directly accessed variable argument. However, if the function's variable arguments are accessed indirectly, the variable arguments will be passed as a pointer to an array allocated in the caller's stack frame.

Once the signatures of all call sites have been determined, we add function arguments that were passed on the stack to the lifted function's signatures. Then, for each coalesced base-pointer set, we allocate variables with the deduced sizes and alignments. Finally, we replace all base pointers with pointers to the newly allocated variables. We ensure we preserve the alignment of these new pointers, if they refer to objects for which we observed they are used in alignment operations. At this point, we can remove the emulated stack from the lifted binary, since all dependencies on the emulated stack were replaced with native stack allocations.

5 Implementation

We implemented WYTIWYG as an extension to BinRec, because it is, to our knowledge, the only dynamic binary to LLVM IR lifter and recompiler capable of translating COTS binaries reliably. We upgraded the LLVM version used by BinRec from 3.8 to 14 and rebased the S2E plugins used for exporting traces onto upstream S2E [5]. Rather than translating the machine code to LLVM IR while the program is running, we use a modified version of RevGen [4] to lift the program offline after completion of the initial tracing. We found this to accelerate the initial tracing drastically and eliminate complexity originating from merging LLVM modules containing the unprocessed traces. Finally, we incorporated a driver that executes tracing, translation and application of refinements automatically.

At the time of development, BinRec did not support lifting of x86 64-bit binaries. Therefore, our prototype targets only x86 32-bit binaries. This does not affect the generality of our approach, because there are no fundamental differences between these two architectures in terms of how the generated code interacts with stack memory.

5.1 Function Recovery

The original version of BinRec did not recover functions and merged all basic blocks into one large function. Naturally, this is an unsuitable representation to lift variables that are local to their function. Hence, we implemented a function recovery similar to the approach detailed in Nucleus [3].

Initially, we create an inter-procedural control-flow graph of the entire binary based on the control transfers that were logged during tracing. Then, we mark any block that is the target of either a direct or indirect call instruction as a function entry. Before function bodies can be computed accurately, all tail-calls have to be identified. Tail-calls are jump-instructions inserted by compilers in place of regular call-instructions. This can happen if a function `f1` calls another function `f2` with the same signature and the call to `f2` would be the very last instruction of `f1` before it would return. Like regular calls, tail-calls can be indirect and/or have a variable number of arguments. The majority of tail-calls can be identified by checking for each direct or indirect jump, whether any of its targets match the entry address of an already identified function.

Sometimes a function has no regular callers and is only encountered as a target of tail calls. Nucleus would merge such functions with their callers and classify the result as a function with multiple entries. Because LLVM IR lacks a natural representation for functions with multiple entries, our implementation splits functions such that there are no overlaps and have only one entry. Our algorithm for this simple: we compute for each function entry the set of blocks reachable through jumps. Then we count in how many functions each block is contained. If a block is contained in more functions than any of its predecessors, it is marked to be a function entry. We found this approach to work reliably across all our inputs, including ones that contain nested and/or indirect tail calls. Functions that are exclusively reachable through a single tail-call and have no regular call sites throughout the entire program are merged with their caller, however. We verified our results by cross-referencing all detected functions with the binary's symbol table (if available) and did not encounter any false positives.

5.2 Variable Argument Library Calls

Since arguments of calls to external functions are passed on the stack, recovering the operands of these calls is a prerequisite to full stack symbolization. If their prototypes are known, BinRec lifts such calls by loading the corresponding arguments from the emulated stack and specifying them as operands to an LLVM call instruction. Fortunately, identifying the arguments of external functions is trivial for most system library functions, because their signatures are known. However, functions that have prototypes with a variable number of arguments, such as `open` or the `printf`-family of functions, require special handling.

To lift calls to these functions, BinRec uses a mechanism called *stack switching*. Because the lifted program pushes all arguments on the emulated stack as required by the external function, the lifted program instructs the *native* stack pointer to point to the emulated stack for the duration of the external call. However, this approach is not compatible with stack symbolization. During symbolization, WYTIWYG eliminates the emulated stack, so it is no longer possible to perform stack switching. Hence, arguments to these calls have to be recovered, *before* WYTIWYG can proceed with symbolization.

There is no uniform way to determine the number of arguments at call sites for variable argument functions. The full prototypes for individual call sites to such functions can usually be determined by inspecting the values of the functions' named arguments at runtime. Therefore, WYTIWYG uses an additional *refinement* before stack symbolization to fully lift calls to these functions. For example, this refinement inspects the format string passed to `printf`-style functions at runtime to determine an exact signature for each call site.

5.3 External Functions

WYTIWYG has to account for any effects on pointers passed to external functions. Because dynamically linked functions are not lifted, we can only instrument calls to them. In our implementation, we maintain a database of known external library functions together with their signatures. The majority of effects necessary for tracking pointers can be expressed through a small set of constraints on the functions' arguments and return values:

- **`ObjectSize`**`(ptr, size, count)`: The object specified by `ptr` is at least as large as the product of `size` and `count` (e.g., `fread`).
- **`ZeroTerminated`**`(ptr)`: The data that `ptr` points to is zero-terminated (such as C strings).
- **`Derive`**`(derived, base)`: The pointer `derived` refers to the same object as the pointer `base` (e.g., `strtok`).
- **`Clear`**`(ptr, [size])`: The external function will clear out any references to stack variables stored in the object that `ptr` refers to (such as structure initialization or `memset`).
- **`Copy`**`(dst, src, [size])`: The external function will copy any references to stack variables stored in the object `src` to the object `dst` (such as structure initialization or `memset`).
- **`FormatStr`**`(str, valist)`: The argument `str` is a C-style format string that describes the arguments contained in a standard C `va_list` (e.g., `vprintf`).

During instrumentation, we translate these constraints into the tracing operations documented in Section 4.2.

6 Evaluation

To evaluate our prototype, we first examine whether our approach retains the functionality of the original binary. We then measure the performance of symbolized binaries to assess whether our approach improves LLVM's ability to reoptimize lifted binaries. Finally, we compare the recovered stack-layouts with the ground-truth provided by the compiler to quantify the accuracy that can be achieved using our approach.

We target the SPECint 2006 benchmark suite, which has been widely used in previous binary lifting and recompilation literature [1, 2, 9, 16]. This benchmark-suite comprises real-world programs, which makes them an ideal target to evaluate the impact binary recompilers have on performance and correctness. We exclude the `omnetpp` and `perlbench`, because our prototype does not handle `setjmp`/`longjmp` and exceptions. We do not evaluate on the SPECfp 2006 set of programs, because x87 instructions are translated using QEMU's software float emulation, and our current implementation does not convert these to LLVM floating-point instructions.

Liu et al. identified BinRec [1], McSema [7], RetDec [12] and mctoll [24] as the best available binary lifters targeting a compiler-level IR [16]. According to their paper, McSema is the only static lifter able to recompile a subset of the SPECint 2006 benchmarks. Although McSema can symbolize stack variables using IDA Pro's stack analyses, the authors admit this process is not automatic because of the heuristic nature of IDA Pro's analyses [8]. For these reasons, we compare WYTIWYG with SecondWrite, which was provided to us by its authors. To our knowledge, it is the only binary to LLVM IR lifter that claims to be capable of recompiling most of the SPECint 2006 benchmarks and supports symbolization of stack variables.

6.1 Functionality

The primary goal of our approach is to recover high-level semantics in binaries without relying on heuristics tailored to the program, compiler or optimization level. To assess whether we achieve this, we compiled each benchmark in multiple configurations. We use the latest GCC 12.2 and Clang 16 compilers at their highest optimization level `-O3`. Additionally, we compiled one set of unoptimized benchmarks with GCC 12.2. SecondWrite could disassemble none of the benchmarks because certain SIMD instructions are not handled by their translator. Hence, we also compiled all benchmarks using GCC 4.4.3 with optimizations enabled on Ubuntu 10.04. This is very close to the GCC version used in the original evaluation of SecondWrite (GCC 4.4.1). We note that, while we did not pass any flags to GCC 12.2 or Clang 16 to emit architecture-specific instructions, older versions of GCC do not emit SSE instructions by default. Since BinRec implements SIMD instructions in software using helper functions provided by QEMU, instruction compatibility is not a concern for WYTIWYG.

We used the `ref` datasets as inputs to trace and validate the recompiled binaries. WYTIWYG successfully lifts and recompiles all binaries and inputs with no manual intervention. Because the `gcc` and `xalancbmk` benchmarks make use of hash maps using pointers as keys, different executions would explore different paths in the lifted binary. We used BinRec's incremental lifting to generate sufficient coverage for these binaries [1]. For the same two benchmarks, we increased the maximal allowed stack-sizes (using `ulimit -s`) due to deeply nested recursive call-chains. WYTIWYG turns tail-calls into regular calls, and the LLVM-signatures of the caller and caller do not always exactly match up in the recovered binary. This prevents LLVM from lowering these calls back to tail-calls.

We recompiled binaries with SecondWrite using default optimizations and disabling speculative disassembly. Without stack splitting, all binaries could be recompiled, except `xalancbmk` and `gobmk`. `xalancbmk` could not be linked and `gobmk` could not be processed by SecondWrite's disassembler. `gcc` crashed on every single `ref` input, even after disabling all of SecondWrite's heuristic optimizations and enabling speculative disassembly. `libquantum` crashed during execution if we enabled stack splitting.

Table 1. Normalized runtime of recompiled binaries relative to the runtime of their respective input binary for each configuration (**SW** = SecondWrite).

		BinRec / WYTIWYG				**SW**
no symbolize	×	GCC 12.2		Clang 16	GCC 4.4	GCC 4.4
symbolize	✓	-O3	-O0	-O3	-O3	-O3
bzip2	×	1.15	0.74	1.21	1.06	0.94
	✓	1.03	0.51	1.13	0.85	0.91
gcc	×	1.39	0.82	1.58	1.18	–
	✓	1.22	0.49	1.25	0.89	–
mcf	×	0.99	0.75	1.09	0.97	0.98
	✓	0.92	0.65	1.07	0.88	1.08
gobmk	×	1.25	0.99	1.20	1.20	–
	✓	0.99	0.79	0.97	0.91	–
hmmer	×	2.38	0.67	1.59	0.72	0.99
	✓	3.04	0.48	1.30	0.60	0.98
sjeng	×	1.06	0.79	1.13	1.09	1.16
	✓	0.85	0.62	0.87	0.82	1.11
libquantum	×	1.15	0.92	1.57	1.16	1.26
	✓	1.21	0.70	1.14	0.89	–
h264ref	×	1.35	0.83	1.60	1.05	1.75
	✓	1.01	0.48	1.23	0.84	1.73
astar	×	0.95	0.69	1.04	0.96	1.08
	✓	0.79	0.47	0.91	0.80	1.08
xalancbmk	×	1.13	0.55	1.23	1.17	–
	✓	0.90	0.10	0.87	0.77	–
Geomean	×	1.24	0.76	1.31	1.05	1.14
	✓	1.10	0.48	1.06	0.82	1.12

We also noticed that SecondWrite cannot lift binaries that have been compiled with position independent code (PIC). It does not handle some types of relocations, that GCC 4.4 emits for position independent code. This is only a minor engineering defect, and we were able to produce a working binary for `mcf` by manually patching these relocations. A more significant issue that we encountered was limited support for jump tables. For example, the jump table in the PIC version of the function `BZ2_decompress` from the binary `bzip2` was entirely missing. Even when enabling speculative disassembly, the jump-targets were not present in the lifted LLVM IR. We assume that SecondWrite cannot identify speculative control transfer targets if the references to them are not encoded as absolute addresses in the binary's data.

6.2 Performance

Our performance experiments were conducted on a system running Ubuntu 22.04 with an AMD Ryzen 9 3900X running at a base clock of 3.8GHz. We disabled frequency boosting, clock-frequency scaling and simultaneous multi-threading to produce consistent results. We instructed LLVM to target the *pentium4* architecture, to avoid measuring the impact of newer CPU features that are available on our target machine.

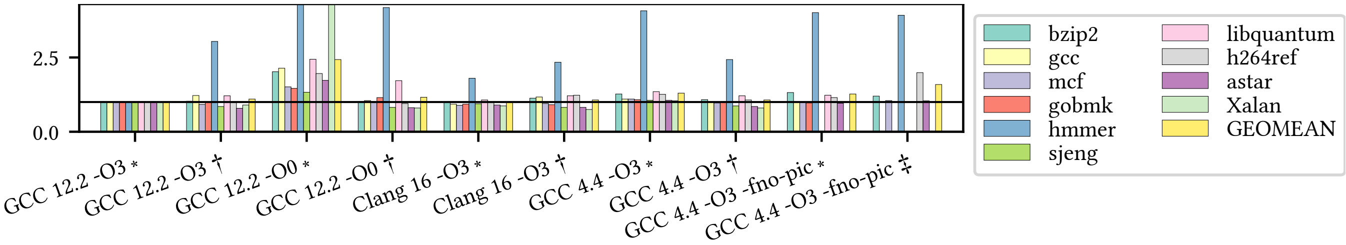


Figure 6. Normalized runtime of input (∗) binaries, binaries recompiled and symbolized with WYTIWYG (†), and binaries recompiled and symbolized with SecondWrite (‡) relative to the runtime of the respective binaries compiled and optimized with GCC 12.2.

Table 1 contains the relative performance impact of recompilation and stack symbolization on each of our input binaries. Across almost all benchmarks, our stack symbolization approach significantly improves the runtime overhead of recompiled binaries, with the worst case average runtime for heavily optimized binaries at 1.10*x*. However, binaries that were not compiled with the latest state-of-the-art compilers can experience a significant uplift in performance: programs compiled with GCC 4.4 see a 1.22*x* speedup, despite being compiled at the highest optimization level. Unoptimized binaries are more than twice as fast, with an average speedup of 2.10*x*. Compared to the non-symbolized versions of those binaries, these performance improvements confirm our hypothesis that recovery of fine-grained stack symbols is central to enhancing the IR-quality of lifted programs and consequently, allows for full-scale program reoptimization.

We also improved the non-symbolized baseline for unoptimized binaries compared to the original version of BinRec from 0.98*x* [1] to 0.76*x*. This is partly due to upgrading BinRec from LLVM 3.8 to LLVM 14, but we found that performing function recovery enhances the results even further. Without function identification, calls were translated into jumps to the function's entry basic block and return instructions were turned into LLVM `switch`-instructions that determine the return target based on the return-address stored on the stack. In the resulting control-flow graphs, frequently called functions act as "chokepoints", because calls appear to return to entirely different call-sites. This optimization hazard is particularly prevalent in unoptimized binaries, where small functions with many call sites are not inlined.

To understand the relationship between recompiled and native binaries, we compared all runtimes in Figure 6 with the baseline of native binaries generated by GCC 12.2. The performance of the recompiled binaries across all `-O3` configurations approaches the GCC 12.2 baseline, although `-O0` is slightly behind. This disparity can be attributed to WYTIWYG not yet recovering global or heap variables, because accesses to these variables are not optimized when compiling a program with `-O0`. Since we do not symbolize these, LLVM's ability to reoptimize these accesses in the lifted program is limited.

Despite symbolization enhancing performance in most cases, there are some outliers: `hmmer` and `libquantum`, when compiled with GCC 12.2 and optimization level `-O3`, experience a degradation in performance. This indicates that LLVM's optimization heuristics are not optimal when applied to the lifted programs. Especially the 2.28*x* (3.04*x* if symbolized) slowdowns of `hmmer` contradict existing binary recompilation literature, where recompiling this benchmark often exhibits one of the greatest performance improvements across SPECint 2006 [1, 2]. However, Figure 6 reveals, that more recent compilers are able to drastically reduce the runtime of this benchmark, to where a 3.04*x* slowdown relative to the GCC 12.2 binary is still faster than the binary produced by GCC 4.4.

We found that vector instructions in the original binaries can cause non-optimal code after lifting. Although LLVM often synthesizes the software-emulated SIMD instructions into LLVM intrinsic vector instructions, the generated sequences are usually more verbose and less efficient. Further, if a function accesses a vector register only partially, it creates a false dependency on the value of that register before the entry of this function. If a program uses SIMD instructions only sparsely (such as `gcc`), these false dependencies can cause vector register values to be copied across multiple function boundaries. Hence, we believe that there is room for improvement by lifting vector instructions more effectively.

Our measurements for SecondWrite diverge with the reported results [2]. Without stack splitting, in the original evaluation, they measured speedups for `libquantum`, `h264ref` and `astar` rather than slowdowns. Similarly, we did not observe a 1.38*x* speedup for `hmmer`. We verified that SecondWrite is compiling the lifted IR with optimizations enabled and could not identify a reason for this disparity. After enabling function splitting, we measured an improvement of 2 percentage points, which appears consistent with the results reported in their paper. We note that SecondWrite does not lift using a separate, emulated stack, and always inlines stack frames as allocations into the lifted LLVM functions. This explains to some extent the smaller improvement compared to the binaries recompiled without stack splitting.

6.3 Splitting Accuracy

To evaluate the accuracy of our approach, we compare the dynamically recovered stack-allocations with the ground-truth generated by LLVM 16's *Stack Frame Layout* analysis. This analysis outputs the stack layout used by code generation for each function when compiling from source. We only consider functions that were executed in our traces, since untraced functions are not contained within the lifted binary. The results are displayed in Figure 7. We assign each ground-truth allocation one of four categories depending on whether it overlaps with an object in the recovered layout: *matched* on perfect match, *oversized*, *undersized* and *missed* on full, partial or no overlap, respectively. Matched and oversized allocations are sufficiently large to prevent overflows, although oversized allocations potentially prevent optimizations. Undersized and missed allocations indicate that there might exist valid inputs to the original program that will cause out-of-bounds accesses in the recompiled program. We note that spilled floating-point and XMM-values can decay into multiple smaller objects due their emulation in software. Although those objects are safely symbolized, our evaluation classifies them as undersized. However, we believe they are only a small fraction of undersized objects. Regardless, our approach achieves a precision of 94.4% and a recall of 87.6% without providing additional inputs.

7 Discussion and Limitations

7.1 Binary Compatibility

Naturally, WYTIWYG can only recover local variables if we can identify functions that have regular stack-frames. Our implementation requires functions to have exactly one entry point, and control transfers between functions to be implemented using `call`- and `ret`-instructions (except for tail calls). The programs are also required to use the stack pointer register and have it point to the bottom of the stack.

Since our approach relies on observing how pointers are used throughout the program, pointer-values need to be "trackable". This means that any operations to derive new pointers from existing ones can be simplified into terms comprising addition and subtraction only. We cannot correctly analyze binaries employing code obfuscation techniques, such as mixed boolean-arithmetic [26], to hide data-flow of pointers.

7.2 Coverage

A primary concern of dynamically driven analyses is attaining comprehensive coverage across the whole program. For WYTIWYG, achieving full coverage encompasses identification of all stack objects and their sizes correctly, and association of all code references with those objects. Albeit our approach yields functional binaries, our evaluation reveals that insufficient coverage leads to function layouts that miss some objects, split them, or assign insufficient space to them.

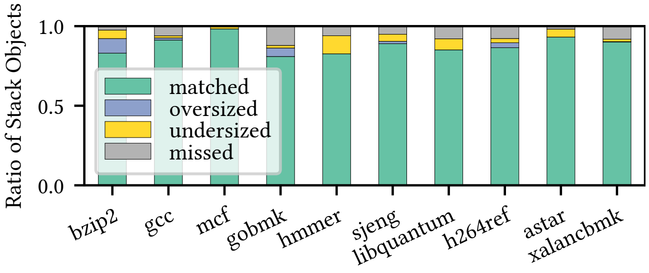


Figure 7. Accuracy of WYTIWYG.

At runtime, this can cause out-of-bound accesses with inputs that were not traced. This affects especially variable-sized stack objects (variable-length arrays and C-style `alloca`) as these are converted into allocations of constant size by our implementation. Although this can be partially remedied by augmenting the binaries with `AddressSanitizer` [19], this incurs a significant performance penalty. For practical purposes, such errors are to be treated as incorrect recompilations.

However, previous work suggests that static approaches are plagued by similar problems. As mentioned in Section 2.2, these approaches operate either conservatively (i.e. splitting only if boundaries are provable) or heuristically (i.e. splitting based on assumption made by developers). Particularly complex functions that would benefit the most from local variable symbolization are also the most difficult to process for these tools. Conservative symbolizers are usually incapable of symbolizing such functions, whereas heuristics will fail eventually and lead to a broken binary with no recourse for fixing except manual intervention.

WYTIWYG provides a path forward in lifting complex programs that exceed the capabilities of static approaches. Using dynamic analysis, complex functions can be symbolized, and we can guarantee that the recompiled program retains the correct functionality for traced inputs. If a new input exhibits invalid behaviours in the recompiled binary, the program can be easily fixed by incrementally reanalyzing it. Further, in the scope of this work, we consider WYTIWYG purely in a vacuum. In practice, our approach could be combined with a robust, heuristics-based static analysis. Such an integration would not only provide the same functional guarantees, but would also minimize issues caused by insufficient coverage.

7.3 Multi-Threading

Because of the inherent difficulties of sound binary recompilation, state-of-the-art binary recompilers have been primarily targeting single-threaded programs. Preserving the memory-orderings between accesses to variables shared across threads poses an additional challenge when lifting multi-threaded binaries.

Usually, synchronization points in programs (e.g., acquisition of a spin-lock) can be identified by seeking special atomic instructions. Since the x86-architecture implements total-store-ordering (TSO), not all synchronizing instructions are marked as atomic, however. TSO provides guarantees about the order in which the effects of non-atomic instructions become visible to other concurrently executing threads. In practice, certain writes to concurrently accessed variables can be implemented with a simple non-atomic store on x86 (e.g., release of a spin-lock). After lifting to IR, LLVM optimizations may reorder non-atomic loads and stores within functions without considering their effect on other threads — possibly breaking original program semantics.

Rocha et al. [18] have shown that by inserting fences strategically around certain memory accesses, the memory-ordering of multi-threaded code can be preserved in lifted programs. Since overuse of fences incurs a significant performance penalty, they employ a heuristic that removes fences around direct memory accesses to the emulated stack frame of the current function. We believe that through local variables symbolization, we can achieve similar results while verifying that variables are used only within a single thread. Since local variable symbolization enhances data-flow analysis results, we might determine that certain stack and heap allocations are never accessed concurrently and safely remove even more fences.

Our analysis itself can be easily adapted to support multi-threading. We would have to extend our runtime to support multiple stacks and make sure that we correctly synchronize concurrent accesses to the data-structures we rely on for stack tracing. However, we cannot easily guarantee that we retain the program's functionality. The non-deterministic nature of multi-threaded programs causes different executions with identical inputs to expose varying behaviours. Exploring all interleavings of threads across these executions is infeasible. Even fixing the scheduling of threads at runtime does not suffice, because simultaneously scheduled threads access shared memory in a non-deterministic order. Since the majority of modern workloads are multi-threaded, we plan to investigate reliable symbolization of such programs in future research.

8 Related Work

Stack Layout Analysis. Most of the work on stack layout analysis focuses on identification of stack variables and their types for decompilation [14, 15, 20, 21, 25]. These approaches aim to aid binary analysis and reverse engineering efforts, and are therefore more tolerant of errors in the recovered information. WYTIWYG instead aims for sound recompilation of lifted binaries. For this use case, probabilistic correctness guarantees are insufficient, as miss-predictions in stack layouts can break semantics of the recompiled binary. A notable exception is SecondWrite [2] which targets recompilation. However, as we demonstrate in our evaluation, SecondWrite's reliance on heuristics limits its applicability to a large set of binaries. WYTIWYG therefore operates only on concrete information collected from execution traces, which allows us to ensure sound stack symbolization and enables our approach to generalize across different binaries, compilers and optimization levels.

Recompilation. WYTIWYG lifts binary code to LLVM-IR in order to enable high-level complex optimization of the lifted IR before lowering the IR back into a semantically correct binary. This use case is also targeted by existing tools like McSema [7], Rev.ng [6], BinRec [1] and SecondWrite [2]. In contrast to WYTIWYG, none of the existing tools (except for SecondWrite) tackle the problem of recovering function stack layouts, which severely limits the compiler's ability to reoptimize the binary. In addition, many approaches [2, 6, 7] rely on unsound heuristics for control-flow recovery. WYTIWYG instead obtains sound control-flow information through dynamic tracing in order to support further IR analysis and transformations. Dynamic approaches, like BinRec [1], try to lift each execution trace within a single step. WYTIWYG instead splits IR generation into multiple phases, where each step refines the IR to make it more suitable for subsequent analysis. Gradually restoring the program's semantics allows us to employ more advanced analyses with reduced complexity, because later analyses can exploit results from the previous steps.

9 Conclusion

This paper proposes WYTIWYG, the first binary recompiler capable of transforming COTS-binaries to a compiler-level IR while recovering stack symbols accurately. By leveraging dynamic analyses to recover source-level structures and iteratively refine lifted binaries, we surpass the limitations of existing recompilers that depend on manually tuned heuristics. Our evaluation demonstrates that the fine-grained partitioning of stack variables allows compilers to reoptimize binaries effectively and achieve considerable speedups compared to previous state-of-the-art solutions. These results highlight the importance of precise stack variable recovery to lifting binaries to an IR that is useful for downstream applications, and that WYTIWYG is highly effective in this regard.

Acknowledgments

We express our gratitude to our reviewers and David Grove, our shepherd, who greatly improved the paper with their valuable remarks.

This material is based upon work partially supported by the Office of Naval Research (ONR) under contracts N00014-22-1-2232 and N00014-21-1-2409, and the Defense Advanced Research Projects Agency (DARPA) under contracts W31P4Q-20-C-0052, N66001-20-C-4027 and 140D04-23-C-0070. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of ONR or DARPA.

References

[1] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. 2020. BinRec: Dynamic Binary Lifting and Recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 36, 16 pages. https://doi.org/10.1145/3342195.3387550

[2] 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* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/2465351.2465380

[3] 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, New York, NY, USA, 177–189. https://doi.org/10.1109/EuroSP.2017.11

[4] Vitaly Chipounov and George Candea. 2011. Enabling sophisticated analyses of x86 binaries with RevGen. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, New York, NY, USA, 211–216. https://doi.org/10.1109/DSNW.2011.5958815

[5] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, CA, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396

[6] 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* (Austin, TX, USA) *(CC 2017)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3033019.3033028

[7] Artem Dinaburg and Andrew Ruef. 2014. McSema: Static Translation of X86 Instructions to LLVM. Presented at *REcon 2014* (Montreal, Canada).

[8] Peter Goodman and Akshay Kumar. 2018. Lifting program binaries with McSema. Presented at *9th International Summer School on Information Security and Protection* (Canberra, AU) *(ISSIP '18)*.

[9] Andrea Gussoni, Alessandro Federico, Pietro Fezzardi, and Giovanni Agosta. 2019. Performance, Correctness, Exceptions: Pick Three. In *Binary Analysis Research Workshop 2019* (San Diego, CA, USA) *(BAR 2019)*. The Internet Society, Reston, VA, USA, 7 pages. https://doi.org/10.14722/bar.2019.23093

[10] R. Nigel Horspool and Nenad Marovac. 1980. An Approach to the Problem of Detranslation of Computer Programs. *Comput. J.* 23, 3 (Aug. 1980), 223–229. https://doi.org/10.1093/comjnl/23.3.223

[11] ISO/IEC. 2020. *ISO/IEC 14882:2020, Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland.

[12] Jakub Křoustek, Peter Matula, and Petr Zemek. 2017. RetDec: An Open-Source Machine-Code Decompiler. Presented at *Botconf 2017* (Montpellier, France).

[13] 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, New York, NY, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[14] JonHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Network and Distributed System Security Symposium 2011* (San Diego, CA, USA) *(NDSS 2011)*. The Internet Society, Reston, VA, USA, 18 pages.

[15] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Network and Distributed System Security Symposium 2010* (San Diego, CA, USA) *(NDSS 2010)*. The Internet Society, Reston, VA, USA, 17 pages.

[16] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. 2022. SoK: Demystifying Binary Lifters Through the Lens of Downstream Applications. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, New York, NY, USA, 1100–1119. https://doi.org/10.1109/SP46214.2022.9833799

[17] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2013. *System V Application Binary Interface — AMD64 Architecture Processor Supplement (Draft Version 0.99.6)*.

[18] Rodrigo C. O. 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* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 888–902. https://doi.org/10.1145/3519939.3523719

[19] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 9 pages.

[20] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, New York, NY, USA, 138–157. https://doi.org/10.1109/SP.2016.17

[21] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Network and Distributed System Security Symposium 2011* (San Diego, CA, USA) *(NDSS 2011)*. The Internet Society, Reston, VA, USA, 18 pages.

[22] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. 2019. From Hack to Elaborate Technique—A Survey on Binary Rewriting. *ACM Comput. Surv.* 52, 3, Article 49 (June 2019), 37 pages. https://doi.org/10.1145/3316415

[23] 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 133–147. https://doi.org/10.1145/3373376.3378470

[24] 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* (Phoenix, AZ, USA) *(LCTES 2019)*. Association for Computing Machinery, New York, NY, USA, 213–218. https://doi.org/10.1145/3316482.3326354

[25] 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, New York, NY, USA, 813–832. https://doi.org/10.1109/SP40001.2021.00051

[26] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson. 2007. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *Information Security Applications*, Sehun Kim, Moti Yung, and Hyung-Woo Lee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–75. https://doi.org/10.1007/978-3-540-77535-5_5