A Compiler-level Intermediate Representation based Binary Analysis and Rewriting System

Kapil Anand Matthew Smithson Khaled Elwazeer Aparna Kotha Jim Gruen
Nathan Giles Rajeev Barua

University of Maryland, College Park
{kapil,msmithso,wazeer,akotha,jgruen,barua}@umd.edu

Abstract

This paper presents component techniques essential for converting executables to a high-level intermediate representation (IR) of an existing compiler. The compiler IR is then employed for three distinct applications: binary rewriting using the compiler's binary back-end, vulnerability detection using source-level symbolic execution, and source-code recovery using the compiler's C backend. Our techniques enable complex high-level transformations not possible in existing binary systems, address a major challenge of input-derived memory addresses in symbolic execution and are the first to enable recovery of a fully functional source-code.

We present techniques to segment the flat address space in an executable containing undifferentiated blocks of memory. We demonstrate the inadequacy of existing variable identification methods for their promotion to symbols and present our methods for symbol promotion. We also present methods to convert the physically addressed stack in an executable (with a stack pointer) to an abstract stack (without a stack pointer). Our methods do not use symbolic, relocation, or debug information since these are usually absent in deployed executables.

We have integrated our techniques with a prototype x86 binary framework called SecondWrite that uses LLVM as IR. The robustness of the framework is demonstrated by handling executables totaling more than a million lines of source-code, produced by two different compilers (gcc and Microsoft Visual Studio compiler), three languages (C, C++, and Fortran), two operating systems (Windows and Linux) and a real world program (Apache server).

Eurosys'13 April 15-17, 2013, Prague, Czech Republic
Copyright © 2013 ACM 978-1-4503-1994-2/13/04…$15.00

1. Introduction

In recent years, there has been a tremendous amount of activity in executable-level research targeting varied applications such as security vulnerability analysis [13, 37], testing [17], and binary optimizations [30, 35]. In spite of a significant overlap in the overall goals of various source-code methods and executable-level techniques, several analyses and sophisticated transformations that are well-understood and implemented in source-level infrastructures have yet to become available in executable frameworks. Many of the executable-level tools suggest new techniques for performing elementary source-level tasks. For example, PLTO [35] proposes a custom alias analysis technique to implement a simple transformation like constant propagation in executables. Similarly, several techniques for detecting security vulnerabilities in source-code [10, 40] remain outside the realm of current executable-level frameworks.

It is a well known fact that a standalone executable without any metadata is less amenable to analysis than the source-code. *However, we believe that one of the prime reasons why current binary frameworks resort to devising new techniques is that these frameworks define their own low-level intermediate representations (IR) which are significantly more constrained than an IR used in a source-code framework*. IR employed in existing binary frameworks lack high level features such as abstract stack and symbols, and are machine dependent in some cases. This severely limits the application of source-level analyses to executables and necessitates new research to make them applicable.

In this work, we convert the executables to the same high-level IR that compilers use, enabling the application of source-level research to executables. We have integrated our techniques in a prototype x86 binary framework called SecondWrite that uses LLVM [27], a widely-used compiler infrastructure, as IR. Our framework has the following main applications:

- **Binary Rewriting** Existing binary backend of the compiler is employed to obtain a new rewritten binary. The presence of a compiler IR provides various advantages

```
int main(){
      int z;
      z = foo(10,20);
      return z;
}
int foo(int a, int b) {
      int temp3,temp1;
      temp1 = a+b;
      if(a>40){
           temp3 = temp1 + 10;
      }
      else {
           temp3 = temp1 - 10;
      }
      return temp3;
}
```
(a) Original C Code

```
char *main(){
     int llvm_tmp3;
     llvm_tmp_3 = rewritten_foo(10,20);
     return llvm_tmp3;
}

int rewritten_foo(int llvm_Arg1,
                           int llvm_Arg2){
     int llvm_tmp4;
     int llvm_tmp2 = llvm_Arg1 +llvm_Arg2;
     if (llvm_Arg1 >  40){
        llvm_tmp4 =  llvm_tmp2 +10;
     }
     else {
        llvm_tmp4 = llvm_tmp2 - 10;
     }
     return llvm_tmp4;
}
```
(d) Recovered C Code with abstract stack and symbol promotion

```
//Global Stack Pointer
int* llvm_ESP;

char *main(){
    llvm_ESP = llvm_ESP-2; //Local Allocation

     llvm_ESP[1] = 20; //Outgoing argument
     llvm_ESP[0] = 10;
     int llvm_tmp_3 = rewritten_foo();
     return llvm_tmp3;
}

int rewritten_foo()
{
     int* llvm_EBP = llvm_ESP;
               //Local Frame Pointer
     llvm_ESP = llvm_ESP-10;
               //Local Allocation

     int tmpIn1 = llvm_EBP[0]; //Incoming Arg
     int tmpIn2; = llvm_EBP[1];

     int  llvm_tmp2 = tmpIn1+tmpIn2;
     llvm_ESP[2] = llvm_tmp2;

     int llvm_tmpIn3 = llvm_EBP[0];
     if (llvm_tmpIn3 > 40){
         int llvm_tmp5 = llvm_ESP[2];
         llvm_ESP[5] = llvm_tmp5 + 10;
     }
     else {
        int llvm_tmp7 = llvm_ESP[2];
        llvm_ESP[5] = llvm_tmp7 - 10;
    }
    int llvm_tmp11 = llvm_ESP[5];
   return llvm_tmp11;
}
```
(b) Recovered C Code with physical stack

```
char *main()
{
    int llvm_ESP2[10];

    llvm_ESP2[1] = 20;
    llvm_ESP2[2] = 10;
    int llvm_tmp1 = llvm_ESP2[1];
    int llvm_tmp2 = llvm_ESP2[2];
    int llvm_tmp_3 =
           rewritten_foo(llvm_tmp2,
                              llvm_tmp1);
    return llvm_tmp3;
}

int rewritten_foo( int llvmArg1,
                    int llvm_Arg2)
{
     int llvm_ESP1[10];

     int llvm_tmp2 = llvm_Arg1+llvm_Arg2;
     llvm_ESP1[2] = llvm_tmp2;

     if (llvm_Arg1 > 40) {
         int llvm_tmp5 = llvm_ESP1[2];
         llvm_ESP1[5] = llvm_tmp5 + 10;
     }
    else {
         int llvm_tmp7 = llvm_ESP1[2];
         llvm_ESP1[5] = llvm_tmp7 - 10;
     }

     int llvm_tmp11 = llvm_ESP1[5];
     return llvm_tmp11;
}
```
(c) Recovered C Code with abstract stack

Figure 1. *Source-code example. Variable names and types in the source-code recovered by LLVM C-backend have been modified for readability.*

to our framework (i) It enables every complex compiler transformations like automatic parallelization and security enforcements to run on executables without any customization. (ii) Sharing the IR with a mature compiler allows leveraging the full set of compiler passes built up over decades by hundreds of developers.

- **Symbolic Execution** KLEE [11], a source-level symbolic execution engine, is employed in our framework without any modifications for detecting vulnerabilities in executables. Our techniques of translating to a compiler IR enable efficient reasoning of input-derived memory addresses using logical solvers without an excessive cost.
- **Source-code recovery** The compiler's C backend is used to convert the IR obtained from a binary to C source-code. The functional correctness of recovered IR ensures the correctness of resulting source-code. Unlike existing tools, which do not ensure functional correctness [7, 23], the source-code recovered by our framework can be updated and recompiled by any source-code compiler.

  Various organizations [3] have critical applications that have been developed for older systems and need to be ported to future versions. In many cases, the application source-code is no longer accessible requiring these applications to continue to run on outdated configurations. The ability of our framework to recover a functionally correct source-code is highly useful in such scenarios.

It is conventional wisdom that static analysis of executables is a very difficult problem, resulting in a plethora of dynamic binary frameworks. However, a static binary framework based on a compiler IR enables applications not possible in any existing tool and our results establish the feasility of this approach for several pragmatic scenarios. We do not claim that we have fully solved all the issues; statically handling every program in the world may still be an elusive goal. However, the resulting experience of expanding the static envelope as much as possible is a hugely valuable contribution to the community.

2. Contributions

We have identified the two tasks below as key for translating binaries to compiler IR. We illustrate the advantages of these two methods through the source-code recovered from a binary corresponding to the example code in Fig 1(a).

• **Deconstruction of physical stack frames** A source program has an abstract stack representation where the local variables are assumed to be present on the stack but their precise stack layout is not specified. In contrast, an executable has a fixed (but not explicitly specified) physical stack layout, which is used for allocating local variables as well as for passing the arguments between procedures.

To recreate a compiler IR, the physical stack must be deconstructed to individual abstract frames, one per procedure.

```
foo(int a, int b) {
    int *p, *q;

    p = &a;
    ...
    *q = ...;
    ... = b;
    }
```
Source Code

```
Stack           q: edx              p: esp + 8
allocations     a: esp + 20         b: esp + 24

foo:
1   subl $16, %esp          // Allocate 16-byte stack frame
2   lea 20(%esp), 8(%esp)   // Put &a(esp+20) into p(esp+8)
3   store ..., (%edx)       // Store to MEM[q]
4   load 8(%esp),%ecx       // Temp ecx ← p (same as &a)
5   load 4(%ecx)            // Load "b" by using the fact that
                               &b = &a + 4 = ecx + 4
```
Pseudo Assembly Code

Figure 2. *A small source-code example and its pseudo-assembly code, showing the limitation of existing methods for detecting arguments.*

```
main() {
    int A[10], i, x;
    x = read-from-file();
    for (i = 0; i < x; i++) {
        A[i] = 10;
    }
}
```

```
main:
1       subl    $48, %esp
2       %ebx = read_from_file
3       mov %ebx, 44(%esp) //Initializing x
4       movl    $0, 40(%esp) //Initializing i
5       jmp     L2  // jump to condition check
    L3:
6       movl    40(%esp), %eax  //load i
7       movl    $10, (%esp,%eax,4) //Reference A[i]
8       addl    $1, 40(%esp)     //Increment i
    L2:
9       cmpl    40(%esp), 44(%esp) //compare x and i
10      jl      L3
```

Figure 3. *An example showing that variable identification and symbol promotion are different.*

Since the relative layout of these frames might change in the rewritten binary, the correct representation requires *all* the arguments (interprocedural accesses through stack pointer) to be recognized and translated to symbols in the IR.

Unfortunately, guaranteeing the static discovery of all the arguments is impossible. Some indirect memory references with run-time-computed addresses might make it impossible for an analysis to statically assign them to a fixed stack location, resulting in undiscovered interprocedural accesses. Existing frameworks circumvent this problem by preserving the monolithic unmodified stack in the IR, resulting in a low-level IR where no local variables can be added or deleted.

Some executable tools analyze statically determinable stack accesses to recognize *most* arguments [5], aiding limited code understanding. However, the lack of guaranteed discovery of *all* the arguments renders such best-effort techniques insufficient for obtaining a functional IR. Fig 2 shows an example procedure where the first argument `a` can be recognized statically while the second argument `b` is not statically discoverable. In the assembly-code, &`a` (`esp+20`) is stored to the memory location for `p` (`esp+8`) (Line 2), which is loaded later to temporary `ecx` (Line 4). The source compiler exploited the layout information (`&a+4=&b`) to load `b` by incrementing `p` (&`a`) by 4 (Line 5). This is safe since the compiler was able to determine that `p` does not alias `q`. However, the executable framework may not be able to establish this relation, since alias analysis in executables is less precise. Hence, it has to conservatively assume that `*q` reference (Line 3) could modify `p` which contained the pointer to `a`. Consequently, the source address at Line 5 is no longer known and argument `b` is not recognized.

Our analysis in Section 4 defines a source-level stack model and checks if the executable conforms to this model. If the model is verified for a procedure, the analysis discovers the arguments statically when possible, but when not possible, embeds run-time checks in IR to maintain the correctness of interprocedural dataflow. Otherwise, stack abstraction is discontinued only in that procedure.

Fig 1(c) demonstrates the impact of abstract stack on the recovered source-code. Fig 1(b) employs a global pointer `llvm_ESP`, corresponding to the physical stack frame in the input binary, for interprocedural communication as well as for representing local allocations in each procedure. However, in Fig 1(c), the stack pointer disappears; instead, local allocations appear as separate local arrays `llvm_ESP1` and `llvm_ESP2` and arguments are represented explicitly.

• **Symbol promotion** Another key challenge we solve is *symbol promotion*, which is the process of safely translating a memory location (or a range of locations) to a symbol in the recovered IR. Existing frameworks do not promote symbols; instead they retain memory locations in their IR [29, 34, 35, 39]. Some post-link time optimizers like Ispike [30] promote memory locations to symbols employing the symbol table information in the object files. However, deployed binaries do not contain symbol information, rendering such solutions unsuitable for our framework.

At first glance, it may seem that the well-known methods for variable identification in executables, such as IDAPro [24] and Divine [6], can be used for symbol promotion. However, this is not the case. The presence of potentially aliasing memory references is a key hindrance to the valid promotion of these identified variables to symbols.

IDAPro characterizes statically determinable stack offsets in the program as local variables while Divine divides the stack memory region into abstract locations by analyzing indirect memory accesses instructions as well.

Fig 3 illustrates the key limitations of both these methods. When the code is compiled, we obtain a stack frame for `main()` of size 48 bytes (10×4 bytes for `A[]`, and 4×2 = 8 bytes for `i` and `x`). The accesses to variables `i` and `x` appear as direct memory references (Lines 3,4,6,9) while the array `A` is accessed using an indirect memory reference (Line 7). Both Divine and IDAPro identify memory locations `esp+44(x)` and `esp+40(i)` as variables based on the direct references. Since the upper bound for the indirect reference `A[i]` is statically indeterminable, even Divine does not generate any useful information about this access. Hence, it creates three abstract locations — two scalars of 4 bytes each, and a leftover range of 40 bytes.

Despite dividing stack memory region into three abstract locations, none of them can be promoted to symbols. It is impossible to statically prove from an executable that the indirect reference at Line 7 does not alias with references to `i` or `x`. Hence, the promotion of memory locations corresponding to `i` and `x` to symbols would be unsafe since it leads to potentially inconsistent dataflow for underlying memory locations. (Source-level alias analyses often assume that any `A[x]` will access `A[]` within its size. However, such size information is not present in a stripped executable.)

Since identification is inadequate for promotion, we have devised a new algorithm to safely promote a set of memory locations to symbols. It computes a set of non-overlapping promotion lifetimes for each memory location taking into consideration the impact of aliasing memory accesses. Our method is oblivious to the underlying method employed for identifying these locations. The locations can be identified by IDAPro, Divine or through a similar method we use.

Fig 1(d) shows the improvement in source-code recovery from symbol promotion, illustrating the replacement of all access to local array `llvm_ESP1` and `llvm_ESP2` in procedures `foo` and `main` respectively by local symbols. As evident, this greatly simplifies the IR and the source-code.

2.1 Benefits of abstract stack and symbols

The presence of abstract stack and symbols has the following advantages:

→ Improved dataflow analysis since standard dataflow analyses only track symbols and not memory locations.

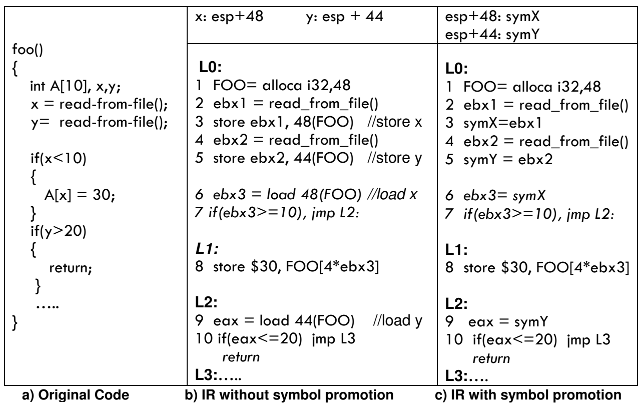
→ Improved readability of the recovered source-code.

→ The ability to employ source-level transformations without any changes. Advanced transformations like compiler-level parallelization [38, 43] add new local variables as barriers and rely on the recognition of induction variables. Several compile-time security mechanisms like StackGuard [19] and ProPolice [21] modify stack layout by placing a *canary* (a memory location) on the stack or by allocating local buffers above other local variables. These methods can be implemented only if the framework supports stack modification and symbol promotion.

→ Efficient reasoning about symbolic memory in case of symbolic execution, as discussed next.

2.1.1 Symbolic Execution

Symbolic execution, e.g. [12], is a well-known technique for automatically detecting bugs and vulnerabilities in a program. Among various challenges facing symbolic execution, handling symbolic memory addresses (addresses derived from user-input) is an important one. There are two primary approaches for handling symbolic memory. Previous symbolic executors for executables [37] make simplifying and unsound assumptions by concretizing the symbolic memory reference to a fixed memory location. On the other hand, popular source-level tools [11, 12] employ logical constraint solvers to reason about possible locations referenced by a symbolic memory operation. Even though the expressions involving symbolic memory become more sophisticated, these tools outperform the former approaches in terms of path exploration and bug detection [13].



a) Original Code
```
foo()
{
   int A[10], x,y;
   x = read-from-file();
   y=  read-from-file();

   if(x<10)
   {
     A[x] = 30;
   }
   if(y>20)
   {
      return;
    }
    .....
}
```

b) IR without symbol promotion
```
x: esp+48        y: esp + 44

 L0:
1  FOO= alloca i32,48
2  ebx1 = read_from_file()
3  store ebx1, 48(FOO)   //store x
4  ebx2 = read_from_file()
5  store ebx2, 44(FOO)   //store y

6  ebx3 = load 48(FOO) //load x
7 if(ebx3>=10), jmp L2:

L1:
8  store $30, FOO[4*ebx3]

L2:
9  eax = load 44(FOO)    //load y
10 if(eax<=20)  jmp L3
      return
L3:.....
```

c) IR with symbol promotion
```
esp+48: symX
esp+44: symY

L0:
1  FOO= alloca i32,48
2  ebx1 = read_from_file()
3  symX=ebx1
4  ebx2 = read_from_file()
5  symY = ebx2

6  ebx3= symX
7  if(ebx3>=10), jmp L2:

L1:
8  store $30, FOO[4*ebx3]

L2:
9   eax = symY
10  if(eax<=20)  jmp L3
      return
L3:....
```

Figure 4. *An example showing the simplification in symbolic execution constraints with symbol promotion.*

The presence of a physical stack and the lack of symbols in an executable pose a difficult challenge in efficiently extending the logical solver based approach for representing symbolic memory in executables. The most straightforward representation of the memory would be a flat byte array. Unfortunately, the constraint solvers employed in existing source-level symbolic execution tools would almost never be able to solve the resulting constraints [11].

```
Constraints:
 A1=write(FOO,48,ebx1)
 A2= write(A1,44,ebx2)
 read(A2,48)<10
 A3 = write(A2,
         4*read(A2,48),30)
Solve:
read(A3,44) <= 20
```

Figure 5. *Constraints for Fig 4(b).*

The segmented memory representation in our framework, obtained by abstract stack and symbol promotion, improves the efficiency of such constraint solvers by enabling them to only consider the constraints related to the segments referenced by the current memory address expression and ignore the remaining segments.

Fig 4 illustrates this case. Fig 4(a) contains a symbolic memory store to array `A`. Fig 4(b) and Fig 4(c) show the pseudo IR obtained from an executable corresponding to Fig 4(a), without and with the application of symbol promotion. Fig 5 shows the constraints and query generated at Line 10 while symbolically executing the path `L0`→`L1`→`L2` in Fig 4(b). Here, `read(A,i)` returns the value at index `i` in array `A` and `write(A,j,v)` returns a new array with same value as `A` at all indices except `j`, where it has value `v`.

However, in Fig 4(c), symbol promotion has segmented the array `FOO` in different segments and references to variables `x` and `y` do not refer the segment `FOO`. Hence, the solver only needs to solve the following simplified query:

$$Solve : symY \leq 20$$

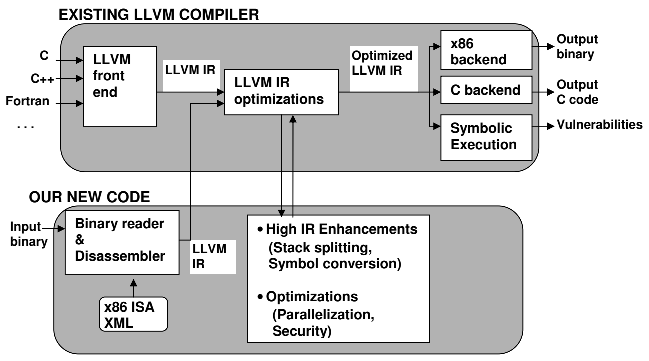


Figure 6. *SecondWrite system.*

This example only shows the simplification of constraints with symbol promotion. The presence of an abstract stack also results in a similar simplification of constraints by segmenting the memory space within each procedure.

3. Overview

Fig 6 presents an overview of the SecondWrite framework. The frontend module, consisting of a disassembler and a custom reader module, processes the individual instructions in an input executable and generates an initial LLVM IR. The framework implements several techniques [20] for recognizing arguments passed through registers and for handling floating point registers. This initial IR is devoid of the desired features like abstract stack frame and symbols. This initial IR is analyzed to obtain an enhanced IR which has all the information and features mentioned previously.

SecondWrite has been already been employed for several applications such as automatic parallelization [26] and security enforcements [31]. As discussed in Section 2, the features of abstract stack and symbols are critical for an efficient implementation of these applications.

3.1 Disassembler Module

The disassembler module implements several mechanisms, as proposed by Smithson and Barua [36], to address code discovery problems and to handle indirect control transfers. Here, we briefly summarize these mechanisms.

A key challenge in executable frameworks is discovering which portions of the code section in an input executable are definitely code. Smithson and Barua [36] proposed *speculative disassembly*, coupled with *binary characterization*, to efficiently address this problem. SecondWrite speculatively disassembles the unknown portions of the code segments as if they are code. However, it also retains the unchanged code segments in the IR to guarantee the correctness of data references in case the disassembled region was actually data.

SecondWrite employs *binary characterization* to limit such unknown portions of code. It leverages the restriction that an indirect control transfer instruction (CTI) requires an absolute address operand, and that these address operands must appear within the code and/or data segments. The code and data segments are scanned for values that lie within the range of the code segment. The resulting values are guaranteed to contain all of the indirect CTI targets.

The indirect CTIs are handled by appropriately translating the original target to the corresponding location in IR through a runtime translator. Each recognized procedure (through speculative disassembly) is initially considered a possible target of the translator, which is pruned further using alias analysis. The arguments for each possible target procedure are unioned to find the set of arguments to be passed to the translator; a stub inside the translator populates the arguments according to the actual target.

The method above is not sufficient for discovering indirect branch targets where addresses are calculated in binary. Hence, various procedure boundary determination techniques, like ending the boundary at beginning of next procedure, are also proposed [36] to limit possible targets.

The disassembler also implements several additional techniques [33] to recover procedure boundaries and inserts additional checks that are essential for the IR to be functional in case of inaccurate recovered boundaries.

3.2 Limitations/Assumptions

SecondWrite relies on the following assumptions, which constitute the limitations of our current framework. We plan to look at them in future work.

- **Disassembly assumptions**: As mentioned above, our underlying disassembler derives possible addresses using the restriction that an indirect control transfer instruction requires an absolute address operand. A compiled code is expected to adhere to this convention unless it has been generated to be position independent.
- **Memory assumptions**: Similar to most executable analysis frameworks [5, 6, 18, 35], our techniques assume that executables follow the *standard compilation model* where each procedure maintains its local frame, which is allocated on entry and deallocated on exit and each variable resides at a fixed offset in its corresponding region. We also assume that in x86 programs, a particular register `esp` refers to the top of memory stack. This assumption is expected to hold in all practical scenarios since x86 ISA inherently makes this assumption. For example, `call` instruction moves `eip` to `esp` and `return` decrements `esp`. An assembly code not adhering to this convention would be extremely hard to write.
- **Self Modifying code**: Like most static binary tools, we do not handle self modifying code. Various tools [41] statically detect the presence of self-modifying code in a program. Such a tool can be integrated in our front-end to warn the user and to discontinue further operation.
- **Obfuscated Code**: We have not tested our techniques against executables with hand-coded assembly or with obfuscated control flow.

4. Deconstruction of physical stack frames

In order to recover a source-level stack representation, we first recognize the local stack frame of a procedure and represent it as a local variable in the IR. As explained in Section 2, this local variable is coupled with the rest of the stack due to interprocedural accesses. We achieve this decoupling by recognizing interprocedural accesses and replacing them with symbolic accesses to the procedure arguments. Below, both these techniques are presented in detail.

4.1 Representing the local stack frame

We begin by finding an expression for the maximum size of the local stack frame in a procedure `X`. We analyze all the instructions which can modify the stack pointer, and find the maximum size, `P`, to which the stack can grow in a single invocation of procedure `X` among all its control-flow paths. `P` need not be a compile-time constant; a run-time expression for `P` suffices when variable-sized stack objects are allowed. An array `ORIG_FRAME` of size `P` is then allocated as a local variable at the entry point of procedure `X` in the IR.

The local variables for the frame pointer and stack pointer are initialized to the beginning of `ORIG_FRAME` at the entry point of procedure `X`. Thereafter, all the stack pointer modifications — by constant or non-constant values — are represented as adjustments of these variables. Allocation of a single array representing the original local frame guarantees the correctness of stack arithmetic inside the procedure `X`.

In some procedures, it might not be possible to obtain a definite expression for the maximum size of the local stack frame. For example, scoped variable-sized local objects in source-code might result in a stack allocation with a non-constant amount, whose expression is not available at the beginning of the procedure. Consequently, a single array `ORIG_FRAME` of a definite size cannot be allocated. Neither can multiple local arrays, one per such stack increment, be allocated since IR optimizations and compiler backend can modify their relative layout thereby invalidating the stack arithmetic. In such procedures, we do not convert the physical stack to an abstract frame. A physical stack frame is maintained in the IR using inline assembly versions of all the stack modification instructions while the remaining instructions are converted to LLVM IR. The runtime checks mechanism presented in the next section is employed to distinguish the local and ancestor accesses.

Persistent stack modification: Returns from a procedure ordinarily restore the value of the stack pointer to the value before the call. However, in some cases, the stack pointer might point to a different location after returning from a procedure call. For example, the called procedure can cleanup the arguments passed through the stack. To represent this stack pointer modification, which persists beyond a procedure call, we introduce the following definition:

Balance Number: The balance number for a procedure is defined as the net shift in the stack pointer from before its entry to after its exit. Four different cases can arise:

Case 1: $\mathit{Balance\ Number} = 0$
This is the common case; no modification required.

Case 2: $\mathit{Balance\ Number} < 0$
This case arises when a procedure cleans up a portion of the caller stack frame and is represented as an adjustment of the stack pointer by *Balance Number* amount in the caller procedure after the call. The amount need not be a constant.

Case 3: $\mathit{Balance\ Number} > 0$
This case implies that a procedure leaves its local frame on the stack and the corresponding frame outlives the activation of its procedure. Such procedures are represented by considering their allocation as part of the caller procedure allocation. The *Balance Number* amount is added to the size of `ORIG_FRAME` array in the caller procedure and the stack pointer is adjusted after the call by this amount.

Case 4: *Balance Number* Indeterminable
In such a case, we do not convert the physical frame into abstract frame and represent the stack as a default global variable in the IR, as shown in Fig 1(b). This is an extremely rare case and in fact, it did not appear in our experiments.

4.2 Representing procedure arguments

As per the source-level representation, we aim to represent all the stack-based interprocedural communication through an explicit argument framework. We discuss why this is not feasible in all the cases and propose our novel methods based on run-time checks to handle such scenarios.

We use Value Set Analysis (VSA) [5] to aid our analysis. VSA determines an over-approximation of the set of memory addresses and integer values that each register and memory location can hold at each program point. Value Set (VS) of the address expression present in a memory access instruction provides a conservative but correct estimate of the possible memory locations accessed by the instruction. VSA accurately captures the stack pointer modifications and the assignments of stack pointer to other registers.

The stack location at the entry point of a procedure is initialized as the base (zero) in VSA and the local frame allocations are taken as negative offsets. Intuitively, memory accesses with positive offsets represent accesses into the parent frame and constitute the arguments to a procedure. A formal argument is defined for each constant offset into the parent frame and each such access is directly replaced by an access to the formal argument.

However, the above method for recognizing arguments is suitable only if VS of the address expression is a singleton set. If the VS has multiple entries, it is not possible to statically replace it with a single argument.

Fig 7 contains an x86 assembly fragment which will be used to illustrate the handling of interprocedural accesses. Fig 8 shows the output IR that results from Fig 7.

```
1. function foo:
2.    sub 100, esp           // Subtract 100 from sp
3.    call bar               // call bar

4. function bar:
5.    sub 10, esp            // Subtract 10 from sp
6.    lea  4(esp),edi        //Move address esp+4 to edi
7.    mov  2, ebx            // Move value 2 to ebx
8.    mov  15, ecx           // Move value 15 to ecx
9.    if (esi < 5) jmp B2    //Conditional Branch

10. B1: mov 4,ebx            // Move value 4 to ebx
11.   mov 16,ecx             //Move value 16 to ecx

12. B2: store 10, ebx[edi]  // Store 10 to indirect offset (edi + ebx)
13.   store 10, ecx[esp]    // Store 10 to indirect offset (esp + ecx)
14.   store 10, edx[edi]    // Store 10 to indirect offset (edi + edx)
```

Figure 7. *A small pseudo-assembly code. The second operand in the instruction is the destination.*

```
1.function foo:
2.    ORIG_FRAME_FOO=alloca i32, 100   // Local frame allocation
3.    call bar(ORIG_FRAME_FOO)         // call bar

4.function bar(i32* inArg)
5.    ORIG_FRAME_BAR=alloca i32, 10    // Local frame allocation
6.    edi = ORIG_FRAME_BAR+4
7.    ebx = 2                          // Move value 2 to ebx
8.    ecx = 15                         // Move value 15 to ecx
9.    if (esi < 5) jmp B2

10. B1: ebx = 4                        // Move value 4 to ebx
11.   ecx = 16                         // Move value 16 to ecx

12. B2: store 10, ebx[edi]             // Store 10 to local frame
13.   store  10, (ecx-SIZE_BAR)[inArg] // Ancestor Store
14.   if ((edx+edi - ORIG_FRAME_BAR) < SIZE_BAR) //Run Time Check
15      store 10, edx[edi]             //Local Store
16.   else
17.     store 10, (edx+edi – SIZE_BAR)[inArg]  //Ancestor Store
```

Figure 8. *IR of the pseudo-assembly code. SIZE_BAR is size of ORIG_FRAME_BAR, register names are pure IR symbols.*

We introduce the following definitions to ease the understanding:

CURRENT_BASE: Stack pointer at the entry point of a procedure.

$\texttt{addr}_\texttt{m}$: The address expression of a memory access instruction m

VS($\texttt{addr}_\texttt{m}$):Value Set of $\texttt{addr}_\texttt{m}$

(x,y): Lower and upper bounds, respectively, of the possible offsets relative to CURRENT_BASE in VS($\texttt{addr}_\texttt{m}$)

LOCAL_SIZE: Size of local frame variable ORIG_FRAME

$\texttt{SIZE}_\texttt{i}$:Size of $\texttt{ORIG\_FRAME}_\texttt{i}$ of the 'ith' ancestor in the call graph, with the caller being represented as the first ancestor. $\texttt{SIZE}_0$ is defined as value 0.

Three different cases for memory reference categorization of a memory access instruction *m* arise:

Case 1: (x,y) ⊂ (-LOCAL_SIZE,0)

This condition implies that the current memory access instruction strictly refers to a local stack location. In Fig 7, Line 12 corresponds to this case. Instruction at Line 6 moves address esp+4 to register edi. Since the size of the current frame in bar (LOCAL_SIZE) is 10 and the local allocations are taken as negative offsets, this translates to VS of edi as {CURRENT_BASE-6}. The VS of ebx at Line 12 is {2,4}; therefore the VS($\texttt{addr}_\texttt{m}$) is {CURRENT_BASE-2,CURRENT_BASE-4}, which translates as a subset of (-LOCAL_SIZE,0). In this case, we replace the indirect access by an access to the local frame as shown Fig 8 (Line 12).

Case 2: $\exists \mathrm{N}: (x,y) \subset (\sum_{i\|i\in(0,N)} \mathrm{SIZE_i}, \sum_{i\|i\in(0,N+1)} \mathrm{SIZE_i})$

This case implies that the current instruction exclusively accesses the local frame of Nth ancestor. In such cases, we make the local frame variable of the Nth ancestor procedure, $\texttt{ORIG\_FRAME}_\texttt{N}$, an extra incoming argument to the current procedure as well as to all the procedures on the call-graph paths from the ancestor to the current procedure. The indirect stack access is replaced by an explicit argument access.

Line 13 in Fig 7 represents this case. Here, VS of ecx is {15,16} which translates to the stack-offset range (5,6) which is subset of $(0,\mathrm{SIZE}_1)$. Line 13 in Fig 8 shows the adjusted offset into the formal argument inArg.

Case 3:

$\exists$ N: { $\{(x,y) \cap(\sum_{i\|i\in(0,N)} \mathrm{SIZE_i}, \sum_{i\|i\in(0,N+1)} \mathrm{SIZE_i}) \neq \emptyset\} \wedge$ $\{(x,y) \not\subset (\sum_{i\|i\in(0,N)} \mathrm{SIZE_i}, \sum_{i\|i\in(0,N+1)} \mathrm{SIZE_i})\}$ }

This case arises when VSA cannot bound the memory access exclusively to the local frame of one ancestor or to the local frame of the current procedure. It also includes cases where VS of the target location is *TOP* (*i.e.*, unknown).

We propose a run-time-check-based solution to represent such accesses in the IR. We define all the possible ancestor stack frames in the call graph as arguments to this procedure. Further, at the indirect stack access, a run-time check is inserted in the IR to dynamically translate the access to the local frame or to one of the ancestor stack frames.

Line 14 in Fig 7 represents this case. Suppose edx is data-dependent and hence its VS is *TOP*. Line 14 in Fig 8 shows the run-time check inserted based on this value. Depending on this check, we either access the local frame (Line 15) or the incoming argument (Line 17).

We have neglected the return address buffer in our calculations for ease of understanding. It is easily considered in our model by adding the return buffer size to each ancestor's local frame size. In the case of dynamically linked libraries (DLLs), the procedure body is not available; hence the above method for handling the arguments cannot be applied. In order to make sure that the external procedures access arguments as before, the LLVM code generator is minimally modified to allocate the abstract frame, ORIG_FRAME, at the bottom of the stack in each procedure in the rewritten binary. Since external procedures are not aware of the call hierarchy inside a program, their interprocedural references are usually limited to only the parent frame. When the prototypes of these external procedures are available (such as for standard library calls), this stack maintenance restriction is avoided altogether by employing the solution presented for any other procedure.

5. Translating memory locations to symbols

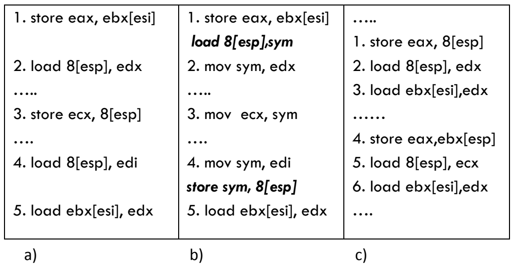
Section 4 presented methods for deconstructing the physical stack frame into individual abstract frames, one per procedure. Even though this representation allows unrestricted modification of the stack frame, accesses to local variables appear as explicit memory references to locations within this array, which are not amenable to standard dataflow analysis. In this section, we propose our methods for translating these memory operations to symbol operations in the IR.

5.1 Motivation for partitions

As discussed in Section 2, maintaining data-flow consistency of the underlying memory locations across the whole program is imperative while promoting memory accesses to symbolic accesses. Fig 9(a) shows a small example with three direct accesses to location `(esp+8)` at Lines 2,3,4; the remaining two are unbounded indirect accesses. The simplest method for maintaining the data-flow consistency across the program is to load the data from the memory location into the symbol just after each aliasing definition, store the symbol back to the memory location just before each aliasing use and promote each candidate stack access to a symbolic access, as shown in Fig 9(b). The load inserted just after the aliasing definition is referred to as a *Promoting Load* and the store just before the aliasing use is referred to as a *Promoting Store* (shown as bold in Fig 9(b)). Although this method ensures correct data flow propagation, it results in a large number of promoting loads and stores which might overshadow the benefit of symbol promotion.

Fig 9(c) illustrates this unprofitable case. In this example, suppose VS of `ebx` is TOP. Consequently, the instructions at Line 3, 4 and 6 are aliasing indirect accesses to the stack location `(esp+8)`. In order to promote the direct memory accesses at instructions 1, 2 and 5, we need to insert Promoting Stores just before instruction 3 and instruction 6 and a Promoting Load just after instruction 4. Hence, promoting three direct memory operations entails the insertion of three extra memory operations, nullifying the benefit.

We propose a novel partition-based symbol promotion algorithm where we divide the program into a set of non-overlapping promotional lifetimes for each memory location. It serves as a fine-grained framework where the symbol



a)	b)	c)
1. store eax, ebx[esi]	1. store eax, ebx[esi]	…..
	load 8[esp],sym	1. store eax, 8[esp]
2. load 8[esp], edx	2. mov sym, edx	2. load 8[esp], edx
…..	…..	3. load ebx[esi],edx
3. store ecx, 8[esp]	3. mov ecx, sym	……
….	….	4. store eax,ebx[esp]
4. load 8[esp], edi	4. mov sym, edi	5. load 8[esp], ecx
	store sym, 8[esp]	6. load ebx[esi],edx
5. load ebx[esi], edx	5. load ebx[esi], edx	….

Figure 9. *Symbol promotion. Second operand in the instruction is the destination of the instruction.*

Statement s	gen[s]	kill[s]
d:store x,mem[reg]	if([sp+addr]∈VS(mem+reg)) d else { }	if([sp+addr]∈VS(mem+reg)) defs(addr) - d else { }
d: store y,addr[sp]	d	defs(addr) - d
d: z = load mem[reg]	{}	{}
d: z = load addr[sp]	{}	{}

Memory location loc : $[sp + addr]$
mem: Non-constant access
$addr$: Constant
$defs(addr)$: Set of instructions defining the memory location [sp+addr]
$in[n]$: Set of definitions that reach the begining of node n
$out[n]$: Set of definitions that reach the end of node n
$pred[n]$: Predecessor nodes of node n
$in[n] = \cup_{i|i \in pred[n]}\{(out[p])\}$
$out[n] = gen[n] \cup (in[n] - kill[n])$

Figure 10. *The reaching definition description. Definitions are propagated across the control flow of program.*

promotion decision can be made independently for each lifetime (a partition) instead of the entire program at once. Not doing symbol promotion in a partition does not affect the correctness of the data-flow in the program. The symbol promotion can be selectively performed in only those partitions where it is provably beneficial. Fig 9(c) shows an intuitive division of the current example into two safe partitions.

5.2 Reaching definition framework

We define a new reaching definition analysis on *memory locations* for computing the partitions. This is different from the standard reaching definitions on *symbols* well-known in compiler theory. For each memory location `loc`, this analysis computes the set of instructions defining the memory location `loc` that reach each program point. The set of definitions includes stores to the memory location `loc` using direct addressing mode as well as possibly aliasing stores.

Fig 10 formulates the reaching definition in terms of VS of the memory accesses. These reaching definitions are propagated across the control flow of the program, similar to the standard compiler dataflow propagation, allowing the partitions to be formed across basic blocks. The interprocedural version of VSA implicitly takes into consideration a local pointer passed to a procedure through an argument.

5.3 Symbol promotion algorithm

The candidates for symbol promotion in a procedure P, represented by a set `LOCS`, are computed as follows:

`M`: Set of memory accesses in `P`

`DM`: Statically determinable memory accesses, $\bigcup_{d \in M}\{d| \|VS(addr_d)\| = 1\}$

`LOCS`: Statically determined stack locations in P, $\bigcup_{d \in DM}\{m|m \in VS(d)\}$

Mathematically, for a stack location `loc`, a single partition constitutes three sets of memory accesses: *DirectAcc*, *BeginSet* and *EndSet*. *DirectAcc* contains statically determinable accesses to the location `loc` and constitutes the potential candidates for symbol promotion. *BeginSet* constitutes the indirect stores that may-alias with `loc` and have a control flow path to at least one element of the set DirectAcc. *EndSet* consists of all the aliasing accesses such that there is

Algorithm 1: *Algorithm for computing partitions for a location* loc *in a procedure P*

```
1  L: Set of loads in P; S: Set of stores in P
2  DL:$\bigcup_{l \in L}\{l|\{loc\} = VS(addr_l)\}$ //Direct Loads
3  IL:$\bigcup_{l \in L}\{l|\{loc\} \subset VS(addr_l)\}$ //Indirect Aliasing Loads
4  DS:$\bigcup_{s \in S}\{s|\{loc\} = VS(addr_s)\}$ //Direct Stores
5  IS:$\bigcup_{s \in S}\{s|\{loc\} \subset VS(addr_s)\}$ //Indirect Aliasing Stores
6  Processed: Set of elements processed
7  while DS != $\emptyset$||IS != $\emptyset$ do
8      define new Partition P, define new list $ActiveList$
9      if DS != $\emptyset$ then
10         s = DS.begin; add s to P.DirectAcc
11     else
12         s = IS.begin; add s to P.BeginSet
13     add s to ActiveList
14     while ActiveList.size!=0 do
15         s = $ActiveList.top$; Add s to Processed
16         for $dl \in DL$ do
17             if $s \in in[dl]$ then
18                 add $dl$ to P.DirectAcc
19                 for $s' \in in[dl]$ do
20                     add s' to $ActiveList$ if s'$\notin$ Processed
21                 remove $dl$ from DL
22         if $s \in IS$ then
23             continue /* No need to store symbol back */
24         for $il \in$ {IL,IS} do
25             if $s \in in[il]$ then
26                 add $il$ to P.EndSet
27             for $s' \in in[il]$ do
28                 add s' to $ActiveList$ if s'$\notin$ Processed
29             remove $il$ from $IL$ if $il \in IL$
```

a control flow path from some element of BeginSet to these accesses. Intuitively, program points just after the elements in BeginSet represent the locations for inserting Promoting Loads. Similarly, program points just before the elements of EndSet are the locations for inserting Promoting Stores.

Algorithm 1 provides a formal description of the method for computing partitions for a memory location `loc`. We begin with an empty partition. We analyze a store instruction, say `ds`. If `ds` is a direct addressing mode instruction then it is added to the DirectAcc set; otherwise it is added to BeginSet (Line 9-12). Load instructions using direct addressing where `ds` is one of the reaching definitions are added to the DirectAcc set of the partition (Line 16-18). The remaining reaching definitions at these load instructions are added to the analysis list (Line 19-20). If `ds` uses a direct addressing mode, indirect load and store instructions with `ds` as one of the reaching definitions are added to the EndSet (Line 24-26). For indirect stores, the symbol need not be stored back to the memory (Line 22-23). As with the direct loads, the rest of the reaching definitions are added to the analysis list (Line 27-29). This analysis is applied repeatedly until the analysis list is empty. At that point, we have one independent partition. We repeatedly obtain new partitions until there are no more direct stores or indirect stores to analyze.

We implement a simple benefit-cost model to determine whether the symbol promotion should be carried out for a particular partition. In a partition, the size of DirectAcc set is the number of memory accesses replaced by symbol accesses. We define $Freq_i$ as the statically determined execu-

App	Source	Lang	LOC	# Proc	Platform
bwaves	Spec2006	F	715	22	Linux
lbm	Spec2006	C	939	30	Linux,Win
equake	OMP2001	C	1607	25	Linux,Win
art	OMP2001	C	1914	32	Linux,Win
wupwise	OMP2001	F	2468	43	Linux
mcf	Spec2006	C	1695	36	Linux,Win
namd	Spec2006	C++	4077	193	Linux,Win
leslie3d	Spec2006	F	3024	32	Linux
libq	Spec2006	C	2743	73	Linux,Win
astar	Spec2006	C++	4377	111	Linux,Win
bzip2	Spec2006	C	5896	51	Linux,Win
milc	Spec2006	C	9784	172	Linux,Win
sjeng	Spec2006	C	10628	121	Linux,Win
sphinx	Spec2006	C	13683	210	Linux,Win
zeusmp	Spec2006	F	19068	68	Linux
omnetpp	Spec2006	C++	20393	3980	Linux,Win
hmmer	Spec2006	C	20973	242	Linux,Win
soplex	Spec2006	C++	28592	1523	Linux,Win
h264	Spec2006	C	36495	462	Linux,Win
cactus	Spec2006	C	60452	962	Linux,Win
gromacs	Spec2006	C/F	65182	674	Linux
dealII	Spec2006	C++	96382	15619	Linux
calculix	Spec2006	C/F	105683	771	Linux
calculix	Spec2006	C/F	105683	771	Linux
povray	Spec2006	C++	108339	3678	Linux,Win
perl	Spec2006	C	126367	2183	Linux,Win
gobmk	Spec2006	C	157883	4188	Linux,Win
gcc	Spec2006	C	236269	6426	Linux,Win
apache server	Real World Program	C	267318	14441	Linux

Table 1. *Benchmarks Table.*

tion frequency at program point i. Hence, the benefit of symbol promotion in terms of eliminated memory references:

$$Benefit = \sum_{i|i \in DirectAcc} \{(Freq_i)\}$$

One promoting load/store is needed for each element of BeginSet and Endset, consequently, the cost:

$$Cost = \sum_{i|i \in BeginSet} \{(Freq_i)\} + \sum_{i|i \in EndSet} \{(Freq_i)\}$$

We calculate the net benefit of each partition as *Benefit - Cost*. Symbol promotion is carried out in a partition only if the net benefit is positive.

6. Results

Table 1 lists all the benchmarks which have been successfully evaluated with the SecondWrite prototype. It includes SPEC2006 benchmark suite, benchmarks from other suites and a real world program, Apache server. Benchmarks on Linux are compiled with gcc v4.4.1 (O0 (No optimization) and O3 (Full optimization) flags) without any symbolic or debug information. Windows benchmarks are compiled with Microsoft Visual Studio compiler (O0 (No optimization) and O2 (Maximum optimization) flags). Only the C and C++ programs are included for Windows since Visual Studio does not compile Fortran. The benchmarks are compiled for x86-32 ISA and results are obtained for SPEC2006 *ref* datasets on a 2.4GHz 8-core Intel Nehalem machine running Ubuntu. The performance analysis of Apache server is carried out using *ab* tool [4]. Analyzing executables compiled by a new compiler causes several engineering challenges with our evolving prototype such as presence of yet unsupported x86 features like SSE and other advanced instructions. However, successful experimentation with distinct compilers such as gcc and Visual Studio demonstrates the lack of any fundamental problem in this regard. In future work, we aim to expand our support base by evaluating executables compiled by other compilers such as Intel compiler and LLVM. Unless mentioned explicitly, the benchmarks in figures are the ones compiled by gcc.

Fig 11 plots the variation in the time taken by SecondWrite, with increasing lines of code, to recover an intermediate representation from an executable. This constitutes the time spent in disassembling the executable and other analyses including abstract stack recovery and symbol promotion. Fig 11 highlights the nearly linear scalability of our framework. The analysis time for large programs such as *gcc*, containing 250,000 lines of code, is around eight minutes. A particular SPEC benchmark *dealII* takes around 35 minutes, forming an outlier to the linear model. It employs templates excessively which causes the compiler to create multiple versions of the same procedure for different template parameters. This extensively slows down several interprocedural analyses resulting in a huge overall analysis time.

6.1 Static characteristics

Our symbol promotion techniques promote the stack memory locations to symbols and direct stack memory accesses to symbol accesses in the IR. On average, 67% of stack locations are promoted to symbols resulting in promotion of 72% of direct stack accesses for the programs listed in Table 1. The detailed statistics for individual benchmarks are presented in an extended version of the paper [1]. For the remaining memory operations, the net benefit for promotion didn't meet the corresponding threshold. Theoretically, our framework can achieve *100% symbol promotion* if the promotion threshold is ignored, but this leads to high overhead in the rewritten binaries due to *Promoting Loads* and *Promoting Stores*. The development of more advanced alias analysis would improve results of our symbol promotion without adversely affecting the performance.

Fig 12 relates the above promoted symbolic references to the original source-level artifacts. We enumerated the symbolic references in the input program using debug information (employed only for counting the references) and compared how many of these symbolic references are restored in the IR. It shows that our techniques are able to restore 66% of the original symbolic references.

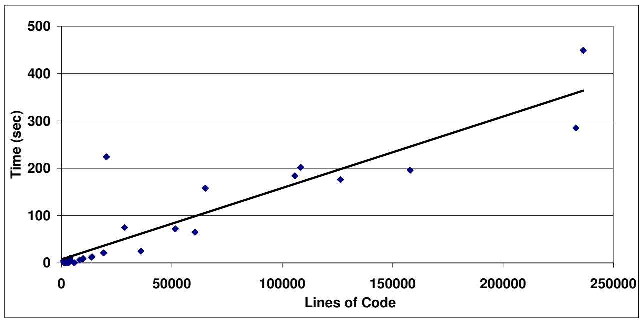


Figure 11. *Variation of analysis time with lines of code. Outlier program* dealII *has been omitted for the ease of presentation.*

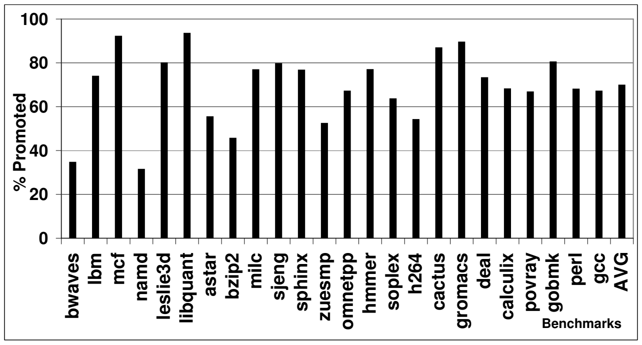


Figure 12. *Percentage of original symbolic accesses recovered in IR.*

Our partitioning algorithm (Alg 1) creates fine-grained promotional lifetimes for each memory location. On average, around 76% of the memory locations have one partition, 18% have two to five, and 6% have five or more partitions. This is not unexpected since large procedures are relatively rare. An extended version of the paper [1] presents the visualization of partitioning results for individual benchmarks.

Table 2 lists the programs which hit corner cases during the deconstruction of physical stack. The analysis of the original source-code revealed that a physical stack frame was required for procedures that call *alloca()*. Runtime checks are inserted in some procedures which accept a variable number of arguments using the *va_arg* mechanism. Most of the procedures using *va_arg* do not require runtime checks. This result establishes our earlier hypothesis

Program	Version	# Proc with Physical stack	# Proc with run-time Checks
gcc	gcc-O0,VS-O0	117	0
gcc	gcc-O3, VS-Ox	117	10
tonto	gcc-O0, gccO3	20	0

Table 2. *Corner cases of our analysis.*

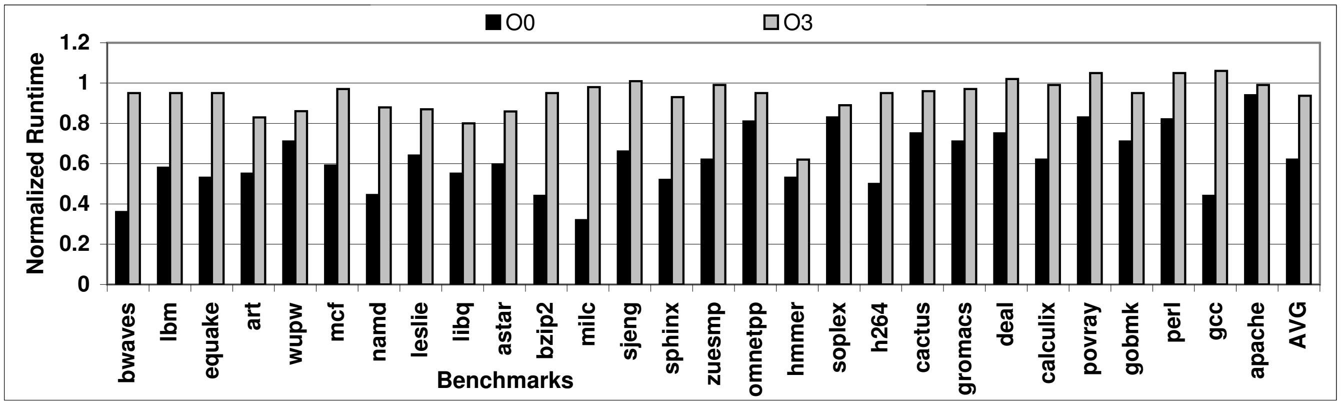


Figure 13. *Normalized runtime of rewritten binary as compared to its corresponding input version (=1.0) compiled by gcc.*

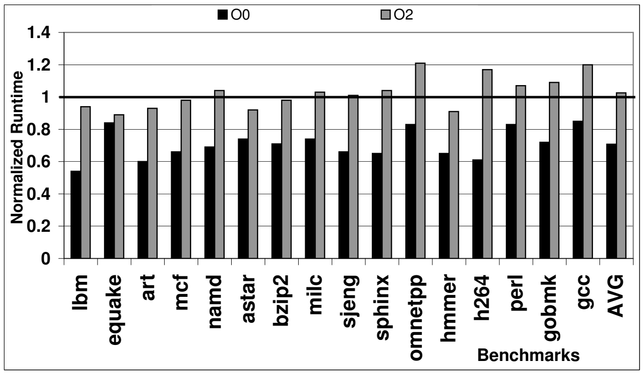


Figure 14. *Normalized runtime of rewritten binary as compared to its corresponding input version (=1.0) compiled by Visual Studio.*

that scenarios requiring run-time checks are extremely rare and consequently, have negligible overhead. Nonetheless, not handling these scenarios prohibits obtaining a functional IR and hence, are imperative for any translation system.

6.2 Un-optimized input binaries

Fig 13 shows the normalized run-time of each rewritten binary compared to an input binary produced using gcc with no optimization (-O0 flag). Fig 14 shows the corresponding run-time for binaries produced using Visual Studio compiler with no optimization (-O0 flag). We obtain an average improvement of 40% in execution time for binaries produced by gcc and 30% for binaries produced by Visual Studio, with an improvement of over 65% in some cases (*bwaves*). In fact, our tool brings down the normalized runtime of unoptimized input binaries from 2.2 to close to the runtime (1.25) of gcc-optimized binaries [1].

6.3 Optimized input binaries

Fig 13 shows the normalized execution time of each rewritten binary compared to an input binary produced using gcc with the highest-available level of optimization (-O3 flag). In this case, we obtain an average improvement of 6.5% in execution time. It is interesting that we were able to obtain this improvement over already optimized binaries without any custom optimization of our own. One of our rewritten binaries (hmmer) had a 38% speedup vs the input binary. Although gcc -O3 is known to produce good code, it missed the creation of few predicated instructions whereas LLVM did this optimization, explaining the speedup. Fig 14 shows the corresponding run-time for binaries produced using Visual Studio compiler with full optimization flag (-O2). As evident, our framework was able to retain the performance of these binaries, with a small overhead of 2.7% on average.

6.4 Impact of symbol promotion

Next, we substantiate the impact of symbol promotion on the run-time of rewritten binaries. Fig 15 and Fig 16 show the normalized improvement in execution time obtained by applying only LLVM optimizations and by applying our symbol promotion techniques. It shows that symbol promotion is responsible for improving the average performance of rewritten binary from 30% to 40% in the case of unoptimized binaries (produced by gcc) and from 1% to 6.5% in the case of optimized binaries (produced by gcc). Since our cost metric is based on static profiling, we observed a small slowdown with symbol promotion in *bzip2 O3*.

It is important to note that these results only measure the impact of symbol promotion. The impact of our method to convert physical frames to abstract frames is not measured above. However, we can infer that number since without obtaining abstract frames, none of the existing LLVM passes would run at all, leading to zero run-time improvement.

6.5 Symbolic Execution

KLEE is efficiently designed to obtain a high code coverage on source programs. We run KLEE in our framework on a set of 50 alphabetically-chosen Coreutils executables and achieve a code coverage of 73% on average compared to 76% obtained by KLEE on source programs when running KLEE with the same options and for the same amount of time (30 minutes/benchmark) in both cases.

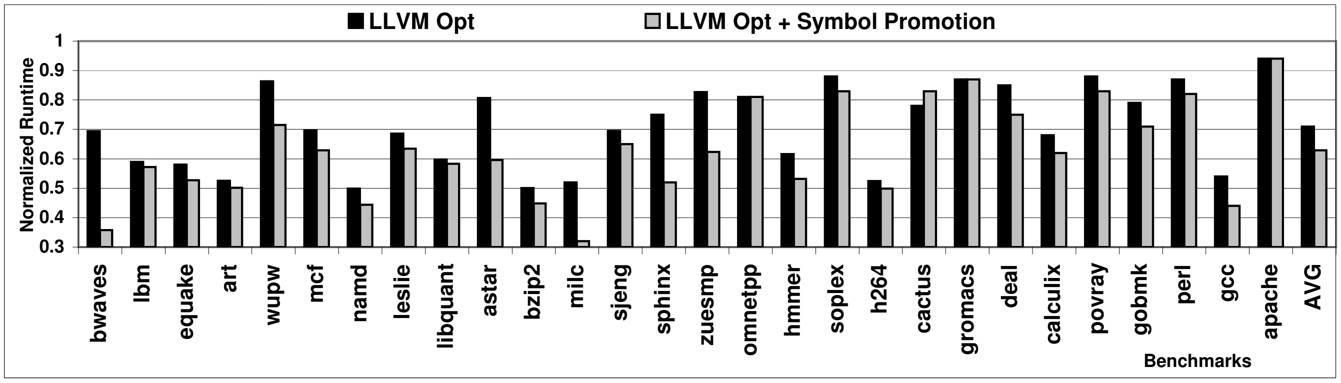


Figure 15. *Impact of symbol promotion on runtime of rewritten binary v/s unoptimized input binary (=1.0).*

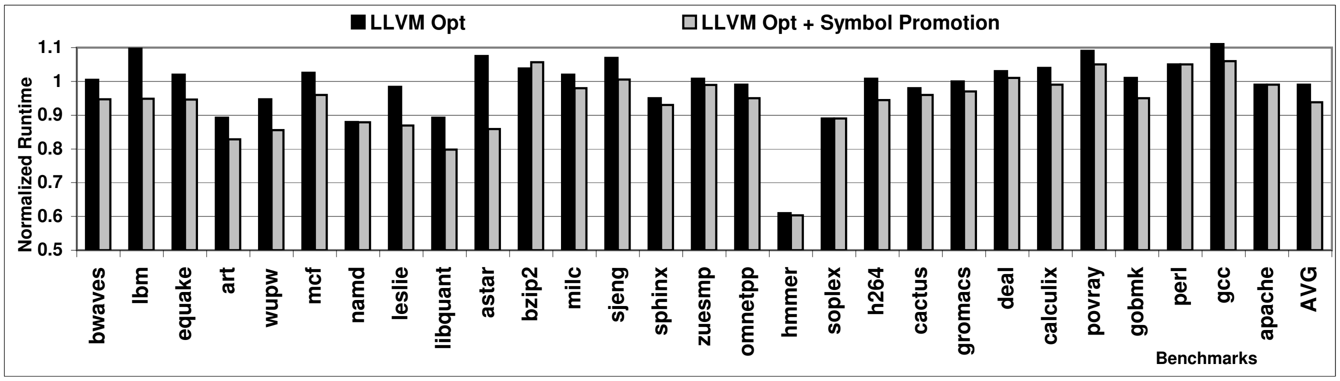


Figure 16. *Impact of symbol promotion on runtime of rewritten binary v/s optimized input binary (=1.0).*

Binary	No Promotion		With Promotion	
	Time(s)	STP Time(s)	Time(s)	STP Time(s)
htget	300	186	37	27
cut	300	252	111	76
split	300	225	157	88

Table 3. *Improvement in constraints processing with symbol promotion.*

Recall from Section 2.1.1 that symbol promotion enables our framework to efficiently reason about symbolic memory accesses. However, most of the Coreutils programs do not contain symbolic array accesses, consequently, these programs are not likely to benefit from our analyses. Instead, a set of programs [13] with known symbolic accesses were chosen to demonstrate the effectiveness of our analysis. Each application was run with KLEE without symbol promotion for five minutes. Then, the applications were run with symbol promotion with the exact same workload. As evident from Table 3, our analysis is highly effective in reducing the time spent by STP solvers in query processing.

KLEE has been shown to detect various bugs in a particular version of Coreutils (6.10). Our framework enables the detection of these bugs from their corresponding executables. Further, the presence of a rewriting path in our framework enables us to remedy the above detected bugs directly from executables. We analyzed the dump for one of the Coreutil executable (*mkdir*), fixed the corresponding behavior in IR and obtained a rewritten bug-free executable.

6.6 Automatic Parallelization

Kotha et al. [26] presented a method for automatic parallelization for binaries. Symbol promotion increases the speedup for a subset of *PolyBench* and *Stream* benchmark suite by 2.25x for 4 threads.

The underlying reason for the speedup is that symbol promotion enables discovery of more induction variables. Many induction variables for outer loops are often allocated on the stack instead of registers. Consequently, such induction variables are not detected by compiler methods, resulting in parallelization of inner loops, which have high synchronization overhead. Symbol promotion enables the discovery of more induction variables, enabling the parallelization of more beneficial outer loops [1].

7. Related Work

Binary rewriting: Binary rewriting research is being carried out in two directions: static and dynamic. None of the previous dynamic rewriters, PIN [29], FX!32 [14], DynamoRIO [8], and others, employ a compiler IR.

Existing static binary rewriters related to our approach include Etch [34], ATOM [22], PLTO [35], Diablo [39] and UQBT [18]. All these rewriters define their own low-level custom IR as opposed to using a compiler IR. These IRs are devoid of features like abstract frames, symbols and maintain memory as a flat address space; the limitations of which have already been discussed in Section 1. PLTO implements stack analysis to determine the use-kill depths of each procedure [35]. However, this information is used only for custom optimizations like load/store forwarding rather than obtaining a high-level IR. UQBT [18] employs procedure prototypes in its IR, but relies on users to provide this information, instead of determining it automatically from an executable. This severely limits its applicability since only the developers have access to that information.

Virtual machines [2] implement stack-walking techniques to determine the calling context by simply iterating over the list of frame pointers maintained as metadata in the dynamic framework; making it orthogonal to our mechanism which statically inserts run-time checks in the IR.

Binary Analysis/Intermediate representation recovery: There are several executable analysis tools such as BAP [9], BitBlaze [37], Phoenix [32] and others which recover an IR from an executable for further analysis. However, these tools have several limitations. All these tools define their own custom IR without the features of abstract stack and symbol promotion, facing limitations similar to tools like Diablo [39] discussed above. Phoenix [32] recovers a register transfer language (RTL) resembling architecture neutral assembly, which does not expose the semantics of several complicated instructions. Further, Phoenix and several other tools require debugging information, which is usually absent in deployed executables.

There are some frameworks which recover LLVM IR from executables. S2E [17] and Revnic [15] present a method for dynamically translating x86 to LLVM using QEMU. Unlike our approach, these methods convert blocks of code to LLVM on the fly which limits the application of LLVM analyses to only one block at a time. Revnic [15] and RevGen [16] recover an IR by merging the translated blocks, but the recovered IR is incomplete and is only valid for current execution; consequently, various whole program analyses will provide incomplete information. Further, the translated code retains all the assumptions of the original binary about the stack layout. They do not provide any methods for obtaining an abstract stack or promoting memory locations to symbols, which are essential for the application of several source-level analyses.

King et al. [25] provide a comprehensive survey of several executable analysis tools. The analysis related to our methods are presented by [5, 6, 42]. Balakrishnan et al. [5, 6] present Value Set Analysis for analyzing memory accesses and extracting high level information like variables and their types. As presented in Section 2, analyzing variables does not guarantee promotion to symbols in IR. Zhang et al. [42] present techniques for recovering parameters and return values from executables but they do not consider the scenarios where the information cannot be derived.

Jianjun et al. [28] promote stack variables to registers dynamically, relying on hardware mechanism for memory disambiguation. In contrast, we provide theoretical formulations for symbol promotion without any hardware support.

Symbolic execution : KLEE [11], EXE [12] are example of source-level tools and cannot be applied directly to executables. Previous binary-only symbolic execution tools like BitBlaze [37] do not represent symbolic memory. S2E [17] and MAYHEM [13] propose a new memory model for simulating symbolic memory, while our techniques enable application of existing models to binaries.

8. Conclusions

This paper presents several component techniques essential for translating executables to a high-level intermediate representation of an existing compiler. Our techniques overcome challenges unique to executables: an explicitly addressed stack, the lack of function prototypes and the lack of symbols. The compiler IR allows the application of source-level complex transformations and advanced symbolic execution strategies on executables and enables functional source-code recovery. In future, we plan to extend our framework for various platform-specific optimizations.

References

[1] A compiler-level intermediate representation based binary system. http://www.ece.umd.edu/∼barua/eurosys13-extended.pdf.

[2] B. Alpern and et. al. The jalapeno virtual machine. *IBM Systems Journal*, 39(1):211 –238, 2000.

[3] Announcement for Binary Executable Transforms. http://www07.grants.gov/.

[4] Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html.

[5] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, pages 5–23. Springer-Verlag, 2004.

[6] G. Balakrishnan and T. Reps. DIVINE: discovering variables in executables. In *Proceedings of the 8th international conference on Verification, model checking, and abstract interpretation*, pages 1–28, 2007.

[7] Boomerang Decompiler. http://boomerang.sourceforge.net/.

[8] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004.

[9] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *CAV*, pages 463–469, 2011.

[10] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, June 2000.

[11] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224, 2008.

[12] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, 2006.

[13] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[14] A. Chernoff and et. al. Fx!32 - a profile-directed binary translator. *IEEE Micro*, 18:56–64, 1998.

[15] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *Proceedings of the 5th European conference on Computer systems*, pages 167–180, 2010.

[16] V. Chipounov and G. Candea. Enabling sophisticated analyses of x86 binaries with RevGen. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 211 –216, 2011.

[17] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 46(3):265–278, 2011.

[18] C. Cifuentes and M. V. Emmerick. UQBT: Adaptable binary translation at low cost. *IEEE Computer*, 33(3):60–66, 2000.

[19] C. Cowan and et al. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium*, pages 63–78, 1998.

[20] K. Elwazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *To Appear in PLDI, 2013*.

[21] H. ETO and K. Yoda. Propolice: Improved stack-smashing attack detection. *IPSJ SIGNotes Computer Security 14 (Oct 26)*, pages 4034–4041, 2001.

[22] A. Eustace and A. Srivastava. ATOM: a flexible interface for building high performance program analysis tools. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference*, pages 25–25, 1995.

[23] Hex-Rays Decompiler. http://www.hex-rays.com/.

[24] IDAPro disassembler. http://www.hex-rays.com/idapro/.

[25] A. King, A. Mycroft, T. Reps, and A. Simon. Analysis of Executables: Benefits and Challenges. *Dagstuhl Reports*, pages 100–116, 2012.

[26] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua. Automatic parallelization in a binary rewriter. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 547–557, 2010.

[27] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–87, 2004.

[28] J. Li, C. Wu, and W.-C. Hsu. Dynamic register promotion of stack variables. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 21–31, 2011.

[29] C.-K. Luk and et al. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM conference on Programming language design and implementation*, pages 190–200, 2005.

[30] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney. Ispike: A post-link optimizer for the Intel Itanium architecture. In *In IEEE/ACM International Symposium on Code Generation and Optimization*, pages 15–26, 2004.

[31] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis. Retrofitting Security in COTS Software with Binary Rewriting. In *SEC*, pages 154–172, 2011.

[32] Phoenix Compiler Infrastructure. http://www.research.microsoft.com/phoenix/.

[33] Recovering function boundaries from executables. http://www.ece.umd.edu/∼barua/function-boundaries.pdf.

[34] T. Romer and et al. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *In Proceedings of the USENIX Windows NT Workshop*, pages 1–1, 1997.

[35] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture. In *In Proc. Workshop on Binary Translation*, 2001.

[36] M. Smithson and R. Barua. Binary Rewriting without Relocation Information. *USPTO patent pending no. 12/785,923*, May 2010.

[37] D. Song and et al. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, pages 1–25, 2008.

[38] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, 2007.

[39] L. van Put, D. Chanet, B. De Bus, B. De Sutler, and K. De Bosschere. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 2005 IEEE International Symposium On Signal Processing And Information Technology*, pages 7–12, 2005.

[40] D. Wagner and et. al. A first step towards automated detection of buffer overrun vulnerabilities. In *In Network and Distributed System Security Symposium*, pages 3–17, 2000.

[41] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. STILL: Exploit code detection via static taint and initialization analyses. In *Computer Security Applications Conference, Annual*, pages 289 –298, 2008.

[42] J. Zhang, R. Zhao, and J. Pang. Parameter and return-value analysis of binary executables. In *Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 501–508, 2007.

[43] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th annual ACM international symposium on Microarchitecture*, pages 85–96, 2002.