Decompilation to Compiler High IR in a binary rewriter

Kapil Anand Matthew Smithson Aparna Kotha Khaled Elwazeer
Rajeev Barua

Technical report, University of Maryland, November 2010

ABSTRACT

A binary rewriter is a piece of software that accepts a binary executable program as input, and produces an improved executable as output. This paper describes the first technique in literature to decompile the input binary into an existing compiler's high-level intermediate form (IR). The compiler's back-end is then used to generate the output binary from the IR. Doing so enables the use of the rich set of compiler analysis and transformation passes available in mature compilers. It also enables binary rewriters to perform complex high-level transformations, such as automatic parallelization, not possible in existing binary rewriters.

Certain characteristics of binary code pose a great challenge while translating a binary to a high-level compiler IR; these include the use of an explicitly addressed stack, lack of function prototypes and the lack of symbols. We present techniques to overcome these challenges. We have built a prototype binary rewriter called SecondWrite that uses LLVM, a widely-used compiler infrastructure, as our intermediate IR, and rewrites both x86 binaries. Our results show that SecondWrite accelerates un-optimized binaries by 27% on average for our benchmarks, and maintains the performance of already optimized binaries without any custom optimizations on our part. We also present two case studies for custom improvement – automatic parallelization and security – to exemplify the benefits and applications of a binary rewriter using a high IR.

1 Introduction

A binary rewriter is a piece of software that accepts a binary executable program as input, and produces an improved executable as output. The output executable typically has the same functionality as the input, but is improved in one or more metrics, such as run-time, energy use, memory use, security, or reliability. In recognition of its potential, binary rewriting has seen much active research over the last decade. The reason for the great interest in research in binary rewriting is that it offers many additional advantages over compiler-produced optimized binaries.

- A rewriter always has the whole program available, and can do **whole program optimization** unconstrained by separate compilation. This allows rewriter to (for example) potentially violate register-usage conventions used by compilers for compatibility during linking.
- Any real compiler misses some optimizations. A rewriter has the ability to **add its own optimizations** to the ones already in the input binary, effectively providing a set union of the optimizations.
- Generic distribution binaries are rarely optimized for any one platform. Further, for compatibility with older processors, commercial binaries rarely use the latest instruction set extensions, *e.g.*, those for multimedia. A rewriter on the end-user's machine can add **platform-specific optimizations** based on that machine's characteristics, specializing the same generic binary differently on each end-user machine.
- A rewriter can apply enhancements to binaries from **any compiler and language** and even when source code is not available, for example to **legacy code, assembly code and third-party code**. It **avoids the need for expensive repeated implementation** of complex code enhancements in multiple compilers.
- A rewriter can retrofit **end-user customized security** to binaries, unlike in compilers where general-purpose security schemes are provided by their vendors.

However the reality of binary rewriters today has fallen far short of this desired vision. Existing rewriters [3, 6, 8, 10, 12, 13, 15, 16] are little more than tools for peephole optimization and instrumentation. Our contention is that the fundamental reason for this limitation is that current rewriters use low-level code representations internally for which it is very difficult to perform complex code transformations. Complex transformations such as extensive whole-program optimizations, automatic parallelization, and sophisticated security enforcement remain outside the capabilities of current rewriters due to low level represention of the intermediate format(IR).

To see why it is so important to represent binaries as high-level IR in rewriters, consider the following characteristics that are needed in any binary rewriting platform to apply unrestricted code transformations:

- **Ability to add and delete any function and any**

call Many dataflow and other optimizations can delete functions and function calls, thus improving run-time. The binary rewriter should allow such deletion for all functions, *not just in some cases,* e.g.*, when a function's prototype can be determined.* It should also be able to add functions and calls to be able to implement enhancements by new code insertion, for example for security or reliability.

- **Ability to add and delete local variables, arguments and return values of functions** Many code optimizations and transformations can delete local variables, arguments and return values resulting in lower run-time; the rewriter should permit this.
- **Ability to run dataflow and alias analysis on all values in the program** Many code optimizations rely on running dataflow and alias analysis on programs. Just like in high level language programs and high IR, such optimizations which are usually run on symbols, must be available on *all* locations in a binary rewriter for it to be an effective platform for optimization.

Unfortunately, existing rewriters cannot provide any of the above three essential capabilities. As a result they are crippled to the point that they are ineffective platforms for general-purpose code transformation. The reasons why existing rewriters cannot provide these capabilities and our technologies for handling each such limitation are outlined below.

- **Inability to split the program stack** In programs, each function has a place in memory called its stack frame. In a binary, the individual stack frame of each function is coupled to that of the caller function and callee functions to form the physical program stack. A program binary code has this *physical stack* with a layout decided but not specified explicitly in the binary. However, a source program (and its IR) has an *abstract stack representation*, which means that local variables are symbolically specified. The program's stack is said to be split when in the IR each constituent stack frame can be represented as separate entity, or is further split into individual variables.
  Challenges:In order to split the stack, a rewriter must both identify the individual components of the stack, and then ensure that no address arithmetic is ever performed across points where the stack is split, which is challenging with indirect stack accesses and unrestricted arithmetic on stack addresses. The use of stack memory locations for passing passing the arguments between functions further exacarbate the problem as indirect stack accesses might perform arithmetic on the current stack pointer to access some ancestor's stack frame. If the stack layout is changed by splitting stack frames, the location accessed is no longer correct.
  Existing rewriters essentially give up on the challenge of stack modification; and instead maintain the stack as an unchanged, monolithic entity [3, 4, 6, 8, 10, 12, 13, 15, 16]. *This has the downside that no new stack variables can be added, none can be deleted, and the layout must be preserved exactly to maintain the correctness of all all arithmetic on the stack pointer and other registers containing pointers to the stack.* In addition, function call deletion is also highly restrictive, and is not possible when a function's prototype is unknown as it might modify stack values in some ancestor function and deletion might remove this side-effect. Further, the addition of new function calls with returns would have to be handled conservatively as stack layout in original function cannot be changed.
  Our solution We first split the stack into frames, one per function and decouple the individual stack frame from the remaining program stack by recognizing function prototypes and represent them symbolically instead of stack accesses. Although many rewriters do not automatically discover function prototypes, some do [6]; we build on that work using sophisticated function prototype recognition. Nevertheless, indirect stack accesses might perform arithmetic on the current stack pointer to access some ancestor's stack frame making the function prototype recognition non-trivial. We propose a solution based on run-time checks for handling such indirect stack accesses by translating them to correct frame at runtime. We further present techniques for splitting each stack frame into individual variables when possible.
- **Inability to translate low-level code to symbols** Whereas high-level languages and compilers reason about locations as "symbols", binaries do not have symbols, but store values in registers and memory locations. However most of compiler theory and optimizations is based on symbols and does not work effectively with memory locations. For example, dataflow and alias analysis track symbols and registers, but do not track (and therefore cannot optimize) values in memory locations.
  Existing rewriters generally represent physical registers and memory locations as such in their internal representation [6,8,10,13,15,16]. In some rewriters [4,12] physical registers are translated to virtual registers, but memory is not converted to symbols. *In all existing binary rewriter, the resulting memory cannot tracked by dataflow and alias analysis, meaning many optimizations are disabled, rendering the rewriters ineffective platforms for optimization.* Custom alias anaalysis and techniques need to be devised [6] to implement even simple optimizations like constant propagation on these memory locations.
  Our solution We have devised techniques by which registers and memory locations in low-level code can

be replaced by symbols in high IR. A key hindrance to legal conversion is the presence of potentially aliasing memory references to the memory locations in question. We use Value Set Analysis [2] for a new purpose: to track potentially aliasing references to allow conversion when possible. When conversion is not possible in for the lifetime of a symbol, we have devised techniques for partial conversion along subsets of the program, which yields the benefits of symbols in at least those parts of the program.

These technologies overcome all the limitations mentioned of existing rewriters, thus enabling binary code to be converted to functional high-level IR of a compiler. We aim to convert the binaries to the same high-level intermediate representation (IR) that compilers use, since compiler IR is well known to support complex transformations. This enables any arbitrary code transformation to easily run on our IR from binaries. Converting binaries to high-level IR is not an end in itself, but acts as a great baseline for applying binary-to-binary optimizations and security enhancements. Moreover, it enables the binary rewriter to leverage the rich set of compiler transformations already present in that compiler.

Our SecondWrite rewriter prototype converts x86 binaries to LLVM IR, and has successfully run the entire standard set of LLVM compiler transformations without any changes to them. The correct rewriting of several SPEC2006 benchmarks provides evidence of the soundness of our techniques. Of course, merely rewriting a highly optimized binary via high-level IR with only standard optimizations is unlikely to yield any improvement in run-time, since the input in already optimized. Our runtime goal in converting to IR is to not run any slower with only standard LLVM, so that the platform is a good basis for the advanced optimizations. In this we succeed: preliminary results show that our basic rewriter rewrites programs via LLVM IR with near break-even performance. The capability of SecondWrite is further demonstrated by implementing existing compiler level affine parallelism yielding speedups and enforcing source-level security enforcements to the binary at low overhead.

2 Advantages of High IR

A compiler high IR based binaryrewriter like Secondwrite has numerous advantages over existing rewriters in terms of its ability to apply transformations and their effectiveness:

- **Ability to do any code transformation** Using high IR allows unrestricted stack modification, necessary for full operation of nearly every compiler transformation.
- **Ability to do effective compiler analysis and optimization** Using high IR allows dataflow and alias analysis to become much more effective. *Platform specific optimizations like register allocation and instruction selection* would not be efficient unless memory locations are converted to symbols.
- **Reuse passes from mature compilers** Mature compiler infrastructures contain a rich set of code analysis and transformation tools. Sharing the high IR from a mature compiler in a binary rewriter allows the rewriter to leverage the full set of passes in the mature compiler.
- **Reuse compiler research** A compiler IR allows rewriter the ability to employ all the existing and future source level analysis and optimization techniques.
- **Program Analysis** Existing compiler backends can be used to convert the obtained compiler IR to source languages like C (e.g. LLVM has a C backend) for better analysis of legacy and other binaries with no source.

Apart from these advantages, characteristics of high IR are imperative for applying certain complex optimizations. Below, we substantiate this claim by depicting how code enhancements in our case studies are infeasible in absence of high IR.

- **Automatic Parallelization** Majority of compiler level parallelization models [19, 23] add new local variables for tasks like barriers and checks, and recognize induction variable symbols. The inability of stack layout modification and lack of dataflow analysis of memory locations in existing rewriters hinders the direct implementation of such source level techniques and absence of enough free registers at required program points might even make them infeasible. On contrast, we can directly employ the source level models and infact, even the same implementation, (as shown in Sec 5) in our rewriter without any changes.
- **Security** Various existing compiler time security defence mechanisms rely on high IR characteristics. For example, Stack Canary Insertion [7] method places a canary (a memory location) on the stack between local variables and the return address. This kind of method can be implemented in a rewriter only if stack modification is allowed. Similarly, ProPolice [11] changes the stack layout to place arrays and other function-local buffers above all other function-local variables requiring function prototypes and ability of stack modification in the intermediate format.

3 Overview

Figure 1 presents an overview of the *SecondWrite* system. The *SecondWrite* system consists of a frontend module for reading the binary executables and generating an initial LLVM IR, a module for extracting more information about the underlying program, optimization passes to implement various optimizations, and the LLVM code generator (Codegen) for obtaining the rewritten binary.

The frontend module consists of a disassembler and a custom reader which processes the individual instructions

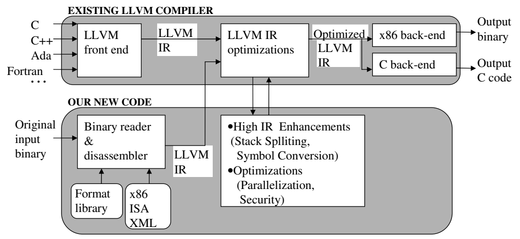


Figure 1: *SecondWrite system*

and generates an initial LLVM IR. [1]. This initial representation is void of the desired IR features like split-stack and symbols. This initial IR is analyzed to obtain an improved IR which has all the information and features mentioned previously. Various optimization passes can be written on the above IR to get an optimized IR. The optimized IR is finally passed to the existing LLVM code generator to obtain the rewritten binary.

4 Decompilation to high-level compiler IR

Abstract and physical stack In programs, stack frame in each function performs four roles: (i) storage of incoming arguments; (ii) storage of local variables, callee saves and register spills; (iii) storage of arguments to any callee functions and caller saves; and (iv) storage of compiler-inserted words like the base pointer and return address. As explained in Sec 1,the program binary code has a *physical stack* with a fixed layout

However a source program (and its IR) has an *abstract stack representation* with local variables symbolically specified. Interprocedural dataflow is represented using an explicit function argument framework where any the incoming arguments to a function are referenced symbolically using formal arguments and actual arguments are exposed as part of its call instruction.

Handling the stack in a rewriter is challenging: first, physical stack layout must be inferred by the rewriter, and second, arithmetic on stack addresses in the binary means that the layout of the stack cannot be changed easily, for example to add new variables, or delete unneeded entries. A way must be found to convert the physical stack in the input binary to an abstract stack in the IR. Moreover, function arguments need to be recognized and represented as symbolic accesses instead of stack accesses.

An obvious way to represent the input binary's stack in the IR would be to represent the entire physical stack as a single global variable in the IR, with another global representing the stack pointer. This (or some similar variant) is how all existing rewriters represent the stack internally. However this has two major drawbacks (i) It requires all stack operations in the binary, such as stack pointer increments and decrements, to be exactly preserved in the IR. This disallows all changes to the stack - no stack variables can be added; and none deleted. (ii) an imprecise representation results since argument based interprocedural dataflow is captured through this global variable, rather than the precise formal and actual arguments in question, hurting the precision of dataflow analysis.

4.1 Splitting the stack into frames

Since a compiler IR has no restrictions while adding or deleting a variable, we try to recover a stack representation similar to what a source program would have generated from the binary. First, we present a technique to recognize the local stack frame of a function and represent it as an independent abstract stack variable. However this frame is still not decoupled from the rest of the stack given layout restrictions. Second, we present a method of decoupling this local stack frame from the program's physical stack by recognizing function argument accesses and return values; and replacing them with symbolic accesses. This decoupling allows the rewriter to freely add any new local variables, arguments, or return values, since there will no longer be any hidden layout restrictions not expressed in the IR.

Representing stack frames We have devised the following method to split the physical stack into individual stack frames, one per function. We begin by finding an expression for the maximum size of the stack frame. To do so, we analyze all the instructions which can modify the stack pointer, and find the maximum limit, P, to which the stack can grow in a single invocation of the current procedure 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. The P-byte space includes all stack frame contents including register spills, arguments, and caller- and callee-save values. An array ORIGINAL_STACK_FRAME of size P is then allocated as a local variable in the IR of the current procedure.

Apart from representing the local stack frame, it is valuable to replace the explicit stack and frame pointers with offsets into the the ORIGINAL_STACK_FRAME array. The need for this is clear – binaries have explicit stack and frame pointers, but high-level programs (and high IR) do not. The elimination of explicit stack pointer avoids unnecessary serialization on its modifications, and in any case, SecondWrite's back-end, like any compiler's back-end, will re-generate a new stack pointer tracking the modified stack. We replace uses of the old stack pointer by offsets into the ORIGINAL_STACK_FRAME array as follows. The offset variables for the frame pointer and stack pointer are initialized to point to the beginning of the ORIGINAL_STACK_FRAME array at start of the function. All stack and frame pointer modification instruc-

[1]Indirect calls and branches occurring in the original binary are handled by translating their target addresses in relocation tables when present in the binary, or through a custom binary rewriting technique we are developing when such tables are absent. However, the choice of technique is orthogonal to the techniques presented in this paper and does not affect our analysis

tions are represented as modifications of this offset variable. Constant propagation converts many of these offsets to constants, or combines many constants into one. Hence, all stack pointer modifications – by constant or non-constant values – are captured exactly through this framework. Thereafter rewriter optimizations can proceed to modify the stack without worrying about correctly maintaining the stack pointer since the resulting code is an abstract stack without an explicit stack pointer.

Recognizing and representing function arguments Binaries employ a combination of architectural registers and stack memory locations for passing the parameters between functions. The register arguments can be determined by direct register dataflow analysis. In this section, we describe the techniques for determining the function arguments passed through memory. We also discuss why the memory arguments cannot be determined in all the cases and propose our methods to handle such scenarios.

Recognizing memory arguments is different for those whose address is a constant offset from the stack pointer (a direct access) versus memory arguments whose address is a non-constant offset in a register (an indirect access). We observe that direct stack memory access instructions with a positive stack offset are arguments of the procedure. To see why, consider that on entry to a procedure, the stack pointer points to the return address and the parameters to the procedure are the locations beyond the return address in the upward direction (assuming a downward growing stack). If the local frame allocations are taken as negative stack offsets, any positive stack offsets represent accesses beyond the local frame and are arguments to the procedure.

The recognition of memory arguments gets complicated due to presence of indirect-addressing mode of memory operands in these architecture. In the indirect addressing mode, the memory address is typically specified as base + index + offset, where base and index are registers. In many cases, the values in these registers cannot be determined statically and depending on the run-time value of these offsets, these possible stack accesses might represent stack memory arguments if they have a positive offset at run-time. Hence, the memory parameter recognition problem becomes non-trivial. Further, in rare highly optimized or hand-coded binaries a function can indirectly access local variables of ancestor functions. Such interprocedural accesses need to be explicitly represented as arguments in a compiler IR

We propose a novel solution to handle the case of indirect memory accesses possibly to the stack. We use Value Set Analysis (VSA) as presented by [2] coupled with our custom solution to handle this problem. VSA is a method to statically compute an over-approximation of the set of values any register or symbol can take. In our case, value set analysis is applied to the non-constant offset expression from the stack pointer, which restricts the range of memory locations the indirect access can take. Indirect accesses are usually used to access the array locations or fields of a structure. Their value set is of form (sp + c + x), Value set analysis would reveal (in most cases) the base address of such access (c in above representation) and the bounds on the value x.

Two different cases can arise. In the first case, (c + upper-bound(x)) is within the bounds of the current function stack frame, which implies that this indirect access is not to an argument but to a local stack location. In second case, (c + upper-bound(x)) lies outside the bounds of current function stack frame. Analysis reveals the ancestor where this address belongs to. We propose to make the bottom of the stack frame of the ancestor function an extra incoming argument to the current function and all functions on call-graph paths from the ancestor to the current function. In this case, the indirect stack access is converted to an explicit memory argument.

Above two cases handle the situation where value set of concerned data object finds a constant base and bounds the remaining unknown offset. However there are still two major limiations. First, VSA, being a best effort analysis, can never guarantee a finite value set for each of these objects. Some objects end up with TOP, specifying that they can point to any value. In fact, no amount of advanced static analysis of the executables can assure the discovery of a bounded range of all indirect access values. Hence, we it is not possible to recognize all the memory arguments. Second, a way has to be found to represent such kind of accesses in IR which can access a local frame or some ancestor frame depending on a statically undeterminable value.

The x86 code example in figure 2 depicts both of the above problems. In this example, a local frame of size 100 bytes is allocated in the entry block of function *foo*. In the following discussion, (sp) refers to the bottom of stack after the above allocation of 100 bytes has been made and $(sp + i)$ refers to i^{th} location from bottom of stack in the upward direction. The pointer to the first argument($(bp - 4)$ or $(sp + 104)$) is moved to the stack location $(sp + 8)$ in the entry block (Instruction 4) and pointer to location $(sp + 40)$ is moved to the same location $(sp + 8)$ in block B3 (Instruction 14). Thus, the stack location $(sp + 8)$ ends up with value set $\{(sp + 40), (sp + 104)\}$. The value in register eax (an input argument) is moved to the local stack location $(sp + 4)$ in the entry block. We assume that argument eax is data-dependant and hence its value set is TOP since it is statically indeterminable. Register ebx is incremented in block B1 and is compared with the value stored at location $(sp + 4)$, which is eax. Since the loop in lines 8-12 iterates an input-dependant number of times, hence the value of ebx, incremented by one in each iteration, is statically indeterminable and also TOP. Load

```
1.  function foo:
2.    Entry:  mov sp, bp         // Create Frame Pointer
3.            sub  100, sp       // Subtract 100 from sp
                                 to allocate local frame
4.            mov bp+4, 8[sp]    // Move pointer to first
                                 memory argument to
                                 local frame location (sp+8)
5.            mov 0, ebx         // Move value 0 to ebx
6.            mov eax, 4[sp]     // Move incoming arg eax to
                                 local frame location (sp+4)
7.            ....
8.    B1:     load 8[sp], edx    // Load from local frame
                                 location (sp+8)
9.            load ebx[edx], ecx// Load from an
                                 unknown stack location
10.           add  1, ebx        // Increment ebx
11.           cmp ebx, [sp]4     // Compare ebx with
                                 local frame value (sp+4);
                                 result in condition flags
12.           bre  B1, B2        // Branch if equal
13.  B2:      ....
14.  B3:      mov bp-60, 8[sp]   // Move a local frame
                                 pointer (sp+40) to local
                                 frame location (sp+8)
15.           br B1
```

Figure 2: *A small assembly code. The second operand in the instruction is the destination of the instruction*

```
1.  Set of Ancestors = {A1,A2,..,An}
        A1 is immediate ancestor
2.  si: Size of stack frames in Ai
3.  A0: Current Function,
4.  s0: size of current stack frame
5.  Si: $\sum$ (sj + retBuf)
6.
7.  d: Representation of Current Indirect Access
8.  NetOffset = d - CurStackBase;
9.  AbsNetOffset = Mod(NetOffset);
                //Absolute value of Netoffset
10.
11. RunTimeCheck:
12. {
13.     if NetOffset $\geq$ 0:
14.         d = d; //Current Frame Access
15.     else if 0$\leq$AbsNetOffSet$\leq$ $S_1$:
16.         d = d[Arg1];//Ancestor 1
17.     else if $S_1$ $\leq$NetOffSet$\leq$ $S_1$:
18.         d = d-$S_1$[Arg2];
19.     ....
20.     else if $S_{n-1}$ $\leq$NetOffSet$\leq$Sn:
21.         d = d-$S_{n-1}$[Argn];
22. }
```

Figure 3: *RunTime check to be inserted in the binary, assuming a downward growing stack*

```
1.  function foo:
2.  Entrymov sp, bp
3.       sub  100, sp
4.       ....
5.       bre  B1,B2
6.  B1:  storeecx, sp[ebx]// Accessing
             an unknown stack location
7.       jump B3
8.  B2:  store eax, 8[sp]
9.       ...
10.      load 8[sp]
11. B3:
```

Figure 4: *A small assembly code. The second operand in the instruction is the destination of the instruction*

instruction 8 accesses data from stack location $(sp + 8)$. The value set of this location implies that destination of load instruction 8 (edx) can point to incoming argument pointer $((sp + 104))$ or a local frame pointer $((sp + 40))$. Thus, at instruction 9, we face two problems. First, it cannot be statically determined whether the memory operand of load instruction 9 lies in the local frame or in the incoming argument space. Second, it creates a problem in determining the prototype of function *foo*.

In such extremely rare cases, we use a fall-back solution with run-time checks. Initially, we make all possible ancestor stack frames in the call graph arguments to this function. Further, at indirect stack accesses where we cannot distinguish whether the accesses are to the current frame or to the arguments, we insert a run-time check in the rewritten binary. This check during the run-time of the output binary whether the access is to the local frame or to one of the ancestor stack frame arguments; thereafter the value is read from the corresponding location.

An interesting observation helps us in limiting the number of ancestors stack arguments in the above case: an indirect access cannot be to any arbitrary ancestor in the call graph. An access to a local variable of an ancestor can be made if there is only one path from ancestor to the current function since in the case of multiple paths, compilers cannot know what is on the stack statically. This limits the ancestors whose arguments need to be passed. In case of above example, the runtime check (shown in pseudo-code format) in figure 3 replaces the load instruction at instruction 9. The overhead of these runtime checks is likely to be very small since we have found these to be extremely rare in our experiments.

4.2 Splitting stack frames into variables

The above section presented methods for splitting the physical stack into individual stack frames, one per function. The individual stack frame in a function behaves like one big local array. However, this representation is still limited since it does not provide individual variables within a frame which can be tracked using dataflow and alias analysis. We propose a two-step analysis for identifying and splitting the individual variables on the stack. Once a stack variable is recognized, the corresponding stack accesses are replaced by accesses to this newly recognized variable; hence improving the quality of IR obtained from a binary.

Stack splitting in presence of direct stack accesses If a function contains only direct stack accesses (accessing address *sp* + *constant-offset*), then the locations at those constant offsets represent different stack variables. A local variable is allocated corresponding to each such stack offset and accesses of type *sp* + *constant-offset* are replaced by accesses to these newly allocated variables. This simple approach is employed by commercial disassembly toolkits like IDAPro for recognizing variables [1].

Stack splitting in presence of indirect stack accesses The above analysis works only if memory is accessed only using direct stack accesses. *Unfortunately even a single indirect stack access renders the above code transformation potentially incorrect since it might possibly alias with direct accesses.* The difficulty is that a indirect stack access to address *sp* + *c* + *x*, where c is constant and x is statically unknown may alias with any stack location since x is unbounded. This prevents the splitting of any direct-accessed stack locations from the frame.

The key intuition for splitting the stack into variables is that if we can bound the range of values for x in indirect stack accesses with address *sp* + *c* + *x*, then that range can be split into an individual variable. The earlier-mentioned Value Set Analysis (VSA) is used to bound the values for x. By repeatedly splitting a stack frame for all indirect accesses in that function, and then splitting direct stack access locations that are not inside the range of any indirect address, a good fine-grained stack splitting into variables results.

4.3 Converting memory accesses to symbols

As explained in section 1, binaries contain data in registers and memory locations. In x86, the shortage of architectural registers yields an especially large number of memory operations. However, much of compiler theory (such as dataflow analysis) is based on symbols and does not work effectively with memory locations. Hence obtaining a IR with symbols is imperative for effectiveness of traditional compiler analysis in a rewriter.

Whole-program symbol conversion Techniques presented earlier split stack frames into individual variables when possible. When a variable is split from the local stack frame, the original accesses to the frame are replaced by accesses to the memory allocated for this new variable. The analysis presented above guarantees that no other memory access aliases with this new memory location; otherwise, the variable would not have been split from the stack at first place. Next, all the registers loaded from, or stored to each variable memory location are identified. Finally, the variable memory location and all its associated registers are simultaneously replaced by a new symbol in the high IR.

Path-only symbol conversion Sometimes a stack or global location cannot be split because it is in range of some indirect access which might alias with it. Here symbol conversion in the whole program is incorrect. However it may be possible to convert the direct-access location to a symbol within a live range where there is no possibly aliasing indirect access. At the beginning of the live range it is unavoidable to load the symbol from memory, and store it to memory at the end of the live range. However in the live range, all memory and register accesses to the location can be converted to symbolic accesses.

Figure 4 shows a code example to illustrate this case. Basic block B1 contains an indirect memory store instruction (Line 6). If the value set of *ebx* is not bounded, then this indirect stack access can possibly alias with any stack access in this function thus prohibiting any stack splitting. However, we notice that path Entry$\rightarrow$ B2$\rightarrow$ B3 is independent of basic block B1. Hence the above indirect stack access instruction cannot affect any memory access instruction in basic block B2. Consequently, the memory access instructions 8 and 10 can be replaced by instruction accessing local variables corresponding to these stack locations, which can be converted to a symbol using the mechanism mentioned above.

To arrive at a formulation for the conversion of memory locations to symbols along certain program regions, let us consider a direct memory access C in function F to memory location m. This location m can be promoted to a symbol in a region in function F around C *if that region has no possibly aliasing memory accesses to m.* To do so, we need to identify all the entry and exit points for the region, and at each entry point load the symbol from memory, and at each exit point, save it back to memory. We call these extra load and stores we insert *promoting loads and stores* respectively. In this way the symbol is in memory outside the region, but not inside the region.

Let us consider how we can find appropriate regions for symbol conversion for reference C. We observe that since a region cannot contain possibly aliasing loads and stores, these aliasing references act as boundaries of the region. However the region might have to made even smaller because of the following constraint for correctness: C must post-dominate all the region entry points. This is because C does not post-dominate an entry point, the promoting load at that entry point might incorrectly convert m to a symbol along path from the entry point to the end of the function that goes outside the region. For an analogous reason, C must dominate all region exit points.

Based on the above, the region entry points are at each of the possibly aliasing loads and stores that are post-dominated by C, and the set of points on the post-dominance frontier such that no path from that point to C contains a possibly aliasing reference. Analogously, the region exit points are at each of the possibly aliasing loads and stores that are dominated by C, and the set of points on the dominance frontier such that no path from C to that point contains a possibly aliasing reference. In this way, the largest legal region for symbol conversion can be defined for every direct access C. The accesses along paths from entry points to C and from C to exit points can be converted to symbols. Hence, mathematically, we can convert those direct accesses which are either post dominated or dominated by C.

Next, we describe our mathematical formulations for this algorithm. A new reaching definition analysis is defined to compute following sets of information at each direct memory access C, to memory location m:

(i) S : The set of possible aliasing stores reaching C

(ii) $\forall s \in S$, Set of direct memory accesses to memory location m in paths from s to C, denoted by DirectAccessFromStore(C,s)

(iii) L : The set of possible aliasing loads reachable from C

(iv) $\forall l \in L$, Set of direct memory accesses to memory location m in paths from the C to l, denoted by DirectAccessToLoad(C,l)

This reaching definition employs VSA for its *gen* and *kill* sets. The details have been omitted due to lack of space. Set S contains all aliasing indirect access which might reach the current direct access, C. For each C, set S is used to determine the set S' of bounding possibly aliasing stores by filtering out the accesses which are not post-dominated by C. The remaining set is denoted by S". Mathematically, set S' and S'' are obtained from set S as follows:

$S' = \{s | s \in S \& IsDominatedBy(s, C)\}$

$S'' = S - S'$

For each element $s'' \in S''$, a postdominator frontier point along path s'' to C is calculated to obtain a set S''_p Similarily, we obtain set L', L'' and L''_p from set L which contains only those elements of L which are dominated by C, remaining elements of L and set of dominator frontier points along path from C to elements of L".

We present a few definitions which aid in understanding the algorithm:

$Freq_i$: Static profiling determined program execution frequency at program point i

L_{Mem} : Memory Access Latency in underlying system

$IsDominatedBy(a, L)$: 1 if L dominates a, 0 otherwise

$IsPostDominatedBy(a, L)$:1 if L postdominates a ,0 otherwise

We implement a simple benefit-cost model to determine whether symbol conversion should be carried out. The total number of direct stack accesses in a range:

$$DirectAccess(C, S, L) = \bigcup \{\{\cup_{s \in S} DirectAccessFromStore(C, s)\} \\ \{\cup_{l \in L} DirectAccessToLoad(C, l)\}\}$$

Conversion of one memory access to symbol results in potential saving of one memory access latency. Hence, the benefit is given by:

$$Benefit = \sum_{i | i \in DirectAccess(C,S,L)} \{(IsPostDominatedBy(i, C) \\ OR \quad IsDominatedBy(i, C)) * (Freq_i) * L_{Mem}\}$$

The insertion of a promoting load and a promoting store constitute the cost and can be calculated as:

$$Cost = (\sum_{l' | l' \in L'} Freq_{l'} + \sum_{p | p \in L''_p} Freq_p \\ + \sum_{s' | s' \in S'} Freq_{s'} + \sum_{p' | p' \in S''_p} Freq_{p'}) * L_{Mem}$$

A net benefit is calculated to find whether to promote the direct stack accesses in current region to symbol by subtracting $Cost$ from $Benefit$. If the net benefit is positive, the current memory location is replaced by a symbol; otherwise the memory accesses are retained as such. This analysis is iteratively performed for all the direct stack accesses in the function till it net benefit becomes negative for all the remaining accesses.

5 Results

A. Rewriting via high IR without enhancements

A subset of *SPEC* and *OpenMP* benchmarks were selected to substantiate the performance of our rewriter. Figure 5 lists the set of benchmarks used for carrying out the experiments. All the benchmarks were compiled with gcc v4.4.1 and results were obtained on a 2.4GHz 8-core Intel Nehalem machine running Ubuntu.

In the first result, all presented binaries ran correctly after rewriting, demonstrating that our techniques to convert to a high IR are sound. We were also able to correctly apply the standard suite of LLVM transformation passes without any changes. Next we consider the runtime performance in two scenarios: when the input binaries were un-optimized, and when they were highly optimized.

Figure 6 shows the normalized run-time of each rewritten binary compared to an input binary produced using gcc with no optimization (-O0 flag). We obtain an average improvement of 27% in execution time on our benchmarks with an improvement of over 34% in some cases. This result shows that our rewriter is useful for binaries that are not highly optimized, such as legacy binaries from older compilers, or binaries from compilers that are inferior compared to the best available compilers.

Next, we study the performance of our rewriter on already optimized binaries. Figure 7 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, the results are mixed, with most benchmarks nearly breaking even or showing a small slowdown, and one benchmark actually showing a speedup of 16%. The average is a 2.7% slowdown.

We consider this near break-even performance on highly optimized binaries a good result for the following two reasons. First, our initial goal was not to get a speedup, but to generate correct IR without introducing too much overhead. This enables the IR to be a starting point for various custom compiler transformations implemented thereafter, such as automatic parallelization or security, as described later. Second, we have currently not implemented any custom serial optimizations that are likely to improve performance further, such as the interprocedural versions of common-sub-expression elimination and loop-invariant code motion, platform specific optimizations and changing compiler-enforced calling convention. We believe these hold promise for speedup with already optimized binaries.

Apart from runtime performance of the rewritten binary, another important metric is metric is the success of memory to symbol conversion. Fig 8 shows the number of direct memory accesses in the original input binary which are replaced by symbols in the IR for each of the benchmarks. The results depicts that on average, 50% of memory accesses are being converted to symbols. This is good considering that only scalar memory accesses can be converted to use symbols instead. Array accesses cannot be converted to symbols (indeed they need not be since they are always memory allocated.)

B. Automatic Parallelization

Application	Source	Lines Of C Source Code
lbm	Spec2006	1155
equake	OMP2001	1607
art	OMP2001	1914
mcf	Spec2006	2685
libquantum	Spec2006	4357
bzip2	Spec2006	8293
sjeng	Spec2006	13847
hmmer	Spec2006	35992
h264ref	Spec2006	51578

Figure 5: Application Characteristics

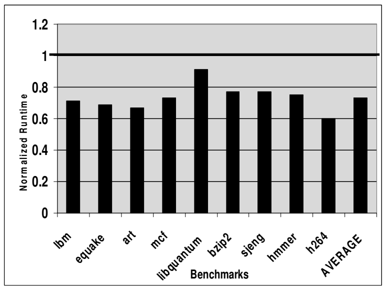


Figure 6: Runtime of rewritten binary v/s unoptimized input binary(=1.0)

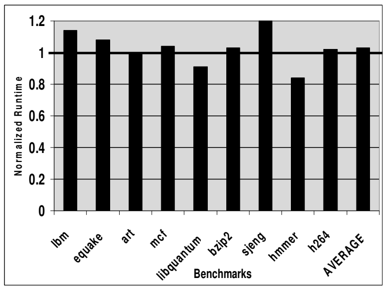


Figure 7: Runtime of rewritten binary v/s optimized input binary(=1.0)

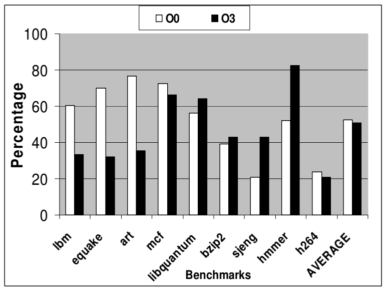
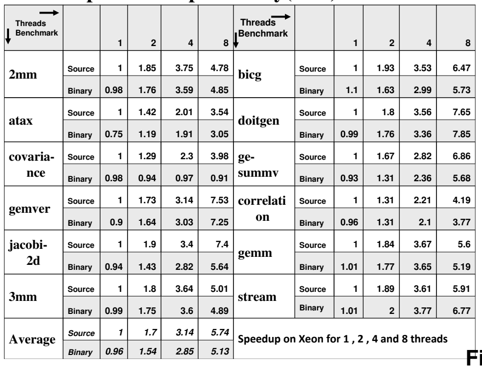


Figure 8: Percentage of direct accesses replaced by symbols



Threads → Benchmark ↓		1	2	4	8	Threads → Benchmark ↓		1	2	4	8
2mm	Source	1	1.85	3.75	4.78	bicg	Source	1	1.93	3.53	6.47
	Binary	0.98	1.76	3.59	4.85		Binary	1.1	1.63	2.99	5.73
atax	Source	1	1.42	2.01	3.54	doitgen	Source	1	1.8	3.56	7.65
	Binary	0.75	1.19	1.91	3.05		Binary	0.99	1.76	3.36	7.85
covaria-nce	Source	1	1.29	2.3	3.98	ge-summv	Source	1	1.67	2.82	6.86
	Binary	0.98	0.94	0.97	0.91		Binary	0.93	1.31	2.36	5.68
gemver	Source	1	1.73	3.14	7.53	correlati on	Source	1	1.31	2.21	4.19
	Binary	0.9	1.64	3.03	7.25		Binary	0.96	1.31	2.1	3.77
jacobi-2d	Source	1	1.9	3.4	7.4	gemm	Source	1	1.84	3.67	5.6
	Binary	0.94	1.43	2.82	5.64		Binary	1.01	1.77	3.65	5.19
3mm	Source	1	1.8	3.64	5.01	stream	Source	1	1.89	3.61	5.91
	Binary	0.99	1.75	3.6	4.89		Binary	1.01	2	3.77	6.77
Average	*Source*	*1*	*1.7*	*3.14*	*5.74*	Speedup on Xeon for 1 , 2 , 4 and 8 threads					
	Binary	*0.96*	*1.54*	*2.85*	*5.13*						

Figure 9: Automatic Parallelization

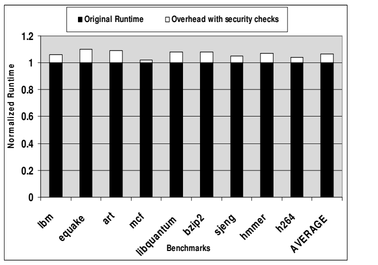


Figure 10: Overhead of security schemes

To illustrate the advantages of binary rewriting using a compiler IR, other researchers [14] have implemented a prototype of an automatic parallelizer inside our rewriter. Automatic parallelization is presented here because of two reasons. First, it is exceedingly important given that most of the software in the world is serial, but most of the hardware is parallel. Second, automatic parallelization is a very challenging compiler transformation relying upon program information not easily available in a binary, such as array bounds, dependence vectors, and alias analysis. These results demonstrate that our platform allows complex transformations (like automatic parallelization) that are infeasible without high IR characteristics.

Our binary rewriter is able to recover enough information for effective parallelization because of its suite of sophisticated static analysis techniques. As evidence of the effectiveness of the parallelizer, we present the speedups when parallelizing source and when parallelizing binaries with increasing number of threads for the Polybench (the Polyhedral Benchmark suite) and Stream(from the HPCC suite) benchmarks in Fig 9. Fig 9 shows the speedup averages 5.7 when parallelizing from source code versus 5.1 when parallelizing from the x86 binary on the x86 Xeon machine with 8 threads. This shows that (a) the speedups are nearly as effective from binaries as from source code and (b) the speedups scale well. It is impressive that the parallelizer also works when compiling to LLVM IR directly from C source code with virtually no changes, and obtains nearly identical speedups.

C. Security Enforcements

Other researchers have suggested schemes which can add security to a binary retro-actively [17] using our high IR based binary rewriter . These schemes defend against all combinations of buffer overflow attacks. Most, if not all, of buffer overflow defenses are source-level defenses require an application program to be re-compiled, which might be no longer availaible in various cases like legacy binaries.

The first security scheme is for protecting the return address on the stack. This is accomplished by maintaining a return address stack in software in a global buffer and comparing this with actual return address at time of return. The next scheme protects against attacks that exploit function pointers. A global variable is declared in the data segment and its address is used as a boundary value for comparing with target of the indirect call. If target is below, the exécuction is halted. Other implemented schemes defend base pointers and *longjmp* buffers.

As evidence of the effectiveness of the implementation of these schemes within a binary rewriter, the eighteen documented attacks on the Wilander and Kamkar benchmarks [21] were run and all were prevented in our rewriter. Figure 10 presents the overhead of the implemented schemes and average overhead introduced is low at 6.5%. Other security checks, such as access permission checks on system calls, are also possible with our rewriter.

6 Related Work

Binary rewriting and link time optimizers have been considered by a number of researchers. Binary rewriting research is being carried out in two directions: static rewriting and dynamic rewriting. Dynamic binary rewriters rewrite the binary during its execution. Examples are PIN [15], BIRD [16], DynInst [13], DynamoRIO [3] and Valgrind [18]. None of the dynamic binary rewriters we found employ compiler IR This is not surprising since dynamic rewriters construct their internal representation at run-time, and hence they would not have the time to construct a compiler IR.

Existing static binary rewriters related to our approach include Etch [8], ATOM [12], PLTO [6] and Diablo [10]. None of the existing binary rewriters employ a compiler level intermediate format; rather they define their own low-level custom intermediate format; the limitations of which have already been discussed in section 1. Diablo defines an augmented whole program control-flow-graph-based IR with program registers as globals and memory as a black box. It does not attempt to obtain high-level information like function prototypes and is geared mainly towards optimizations like code compaction. Taking memory as a black box limits its applicability to architectures like x86 which have a limited set of registers. ATOM defines a symbolic RTL-based intermediate format with infinite registers but does not do any attempt of analyzing or modifying the stack layout. Its mainly targeted towards RISC architectures like Compaq Alpha. PLTO employs a whole program CFG based IR and implements stack analysis to determine the use-kill depths of each function [5]. However this information is not used for converting it into high-level IR; rather it is used only for low-level custom optimizations like load/store forwarding. Etch does not explicitly build an IR and allows user to add new tools to analyze binaries. The primary goal of Etch appears to be instrumentation and has only been shown to be applicable for simple optimizations like profile-guided code layout.

UQBT [4] is a binary translation framework which defines its own custom IR as opposed to using an existing compiler's IR; hence it loses out on the advanced set of optimizations implemented in an already existing mature compiler infrastructure. The IR involved is high level involving procedure prototype abstraction but the conversion to IR relies on user-provided information about the number of parameters and their locations, instead of determining that information automatically from a binary like we do. This severely limits the applicability of UBQT since only the developers have access to that information, and moreover, the translation process to an intermediate form is no longer automatic.

Regarding the analysis of executables, Ramakrishnan et al [20] suggest static analysis of binaries to check whether coding standards have been followed. Gogul et al [2, 9] present value set analysis for analyzing memory accesses and extracted high level information like variables and their types. As presented in section 4, analyzing variables does not guarantee conversion to symbols. Further, their prime aim is to employ this information for security analysis of executables and not for the rewriting of executables. Zhang et al [22] present techniques for recovering parameters and return values from executables but they do not consider situations where the information cannot be derived, specially in presence of indirect memory accesses. As mentioned before, such best effort solutions are good for executable analysis but do not certify the reliable behavior once these analyses fail. Further all these analysis methods have neither been used for the generation of compiler IR nor for any rewriting purposes.

7 Conclusions

In this paper, we present SecondWrite, our binary rewriting tool which decompiles the input binary into LLVM IR and then uses the LLVM compiler's back-end to generate the output binary from the IR. Doing so enables the use of the rich set of compiler analysis passes already available in LLVM and allows SecondWrite to be applied for complex and high-level custom compiler transformations like automatic parallelization and security enforcement. In future, we plan to develop custom serial optimizations targeted towards binaries and implement more sophisticated automatic parallelization techniques.

References

[1] Idapro disassembler. http://www.hex-rays.com/idapro/.

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

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

[4] C. Cifuentes and M. Van Emmerick. Uqbt: Adaptable binary translation at low cost. *IEEE Computer*, 33(3):60–66, March 2000.

[5] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–24, New York, NY, USA, 1998. ACM.

[6] Benjamin Schwarz et al. PLTO: A link-time optimizer for the intel ia-32 architecture. In *In Proc. 2001 Workshop on Binary Translation*, 2001.

[7] Cowan et al. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, pages 63–78, Berkeley, CA, USA, 1998.

[8] Ted Romer et al. Instrumentation and optimization of win32/intel executables using Etch. USENIX Windows NT Workshop, August 1997.

[9] Thomas Reps et al. Intermediate-representation recovery from low-level code. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 100–111, New York, NY, USA, 2006. ACM.

[10] Van Put et al. 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, Athens, December 2005. IEEE.

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

[12] Alan Eustace and Amitabh 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, Berkeley, CA, USA, 1995. USENIX Association.

[13] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. Scalable High Performance Computing Conference, May 1994.

[14] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua. Automatic parallelization in a binary rewriter. In *Proceedings of the International Symposium on Microarchitecture 2010*. ACM Press.

[15] Chi-Keung et al Luk. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[16] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 358–370, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Retrofitting security to COTS binaries. http://www-ece.umd.edu/˜barua/security-techreport.pdf/.

[18] J. Seward and N. Nethercote. Valgrind, an open-source memory debugger for x86-linux. http://developer.kde.org/˜sewardj/.

[19] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[20] Ramakrishnan Venkitaraman and Gopal Gupta. Static program analysis of embedded executable assembly code. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 157–166, New York, NY, USA, 2004. ACM.

[21] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162. Citeseer, 2003.

[22] Jingbo Zhang, Rongcai Zhao, and Jianmin Pang. Parameter and return-value analysis of binary executables. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 501–508, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Craig Zilles and Gurindar Sohi. Master/slave speculative parallelization. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 85–96, Los Alamitos, CA, USA, 2002.