Scalable Variable and Data Type Detection in a Binary Rewriter

Khaled ElWazeer Kapil Anand Aparna Kotha Matthew Smithson Rajeev Barua
Electrical and Computer Engineering Department, University of Maryland College Park, MD, 20742, USA
{wazeer,kapil,akotha,msmithso,barua}@umd.edu

Abstract

We present scalable static analyses to recover variables, data types, and function prototypes from stripped x86 executables (without symbol or debug information) and obtain a functional intermediate representation (IR) for analysis and rewriting purposes. Our techniques on average run 352X faster than current techniques and still have the same precision. This enables analyzing executables as large as millions of instructions in minutes which is not possible using existing techniques. Our techniques can recover variables allocated to the floating point stack unlike current techniques. We have integrated our techniques to obtain a compiler level IR that works correctly if recompiled and produces the same output as the input executable. We demonstrate scalability, precision and correctness of our proposed techniques by evaluating them on the complete SPEC2006 benchmarks suite.

Categories and Subject Descriptors D.2.7: Software Engineering [*Distribution, Maintenance, and Enhancement*]: Restructuring, reverse engineering, and reengineering

Keywords reverse engineering; binary rewriting; variable recovery; type recovery;

1. Introduction

Reverse engineering binary executable code is commonplace today, especially for untrusted code and malware. Agencies as diverse as anti-virus companies, security consultants, code forensics consultants, law-enforcement agencies and national security agencies routinely try to understand binary code. Existing tools such as the IDAPro disassembler and the Hex-Rays decompiler [1] help, with the latter producing (non-executable) C-like pseudocode text.

However, existing reverse engineering tools do not exhibit several desired characteristics. First, previous tools do not aim to recover a *fully-functional high-level code* (similar to source code) from executables. These tools neglect variables allocated on the floating point stack and generate intermediate representation (IR) containing incomplete interprocedural interfaces. The recovered IR is suitable for human understanding but does not capture the complete functionality of the input executable. Second, they are either imprecise [1] or recover precise information at the cost of *scalability*. For example, DIVINE [5], the most precise variable identification tool proposed in the literature, spends two hours while analyzing programs of the order of 55,000 assembly instructions.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.
Copyright © 2013 ACM 978-1-4503-2014-6/13/06. . . $15.00

Recovering a functional IR in a scalable and accurate manner would be invaluable to security professionals. It would enable them to write compiler passes to extract properties of interest. The recovered IR can be updated with insertion, deletion, or modification. Running the updated rewritten program enables dynamic source-level debugging techniques such as judiciously placed print statements, and many more.

Recovering functional IR is also valuable for legacy binaries for which the source code has been lost. It enables users to fix bugs in such binaries, modify the functionality, optimize such binaries, or even port them to new hardware systems.

In this work, we present static analyses that can recover source level variable and type information from x86 binaries as large as millions of instructions in a few minutes. The produced information is as accurate as the current state of the art x86 binary analysis systems. The recovered information is represented in a high level compiler IR that is completely functional and produces a correct rewritten executable when recompiled. Our static techniques combine functionality, precision and scalability; features that collectively do not exist in today's binary analysis tools.

Our methods also improve the scope of variable analysis and type recovery in two ways. First, unlike current binary analysis techniques, our recovery mechanisms are able to recognize variables allocated on the floating point stack. Recognizing such variables is a hard problem in the presence of unknown indirect and external calls. Recognizing floating-point stack variables is imperative for obtaining a functional IR from executables.

Another way our methods improve the scope of analysis is by accurately identifying all register allocated variables used as arguments and returns from functions. Most x86 binary analyses will only identify memory allocated arguments [4], but do not identify register allocated ones. This is acceptable when recovering pseudocode, but unacceptable when recovering functional code. Some methods perform either brute force techniques [22] that are imprecise, or use dynamic analysis to detect arguments and returns [7] which is precise but produces incomplete information.

This work presents a step towards a system that rewrites executables into a functional high-level program representation and incorporates as much source level information as possible in a scalable manner. We envision the need for such a system in various security and binary analysis applications. This work has the following contributions:

* It produces a correct and running IR that can be recompiled to obtain a rewritten executable that works exactly the same way as the input executable.

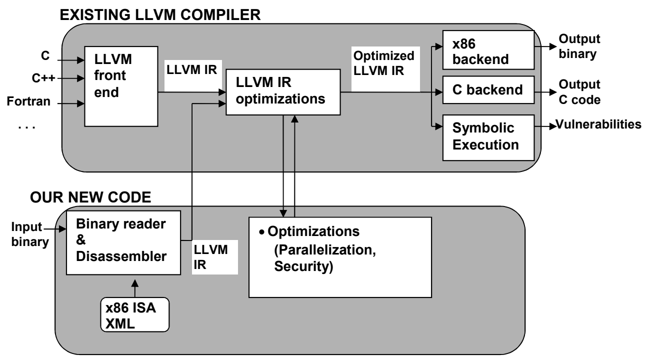

Figure 1. SecondWrite Flow

- ∗ It presents algorithms for solving problems missed while analyzing executables like resolving floating point stack accesses and accurately identifying interprocedural interfaces.
- ∗ It presents a highly scalable mechanism for identifying variables and types which is orders of magnitude faster than current analysis techniques. Our techniques do not rely on symbol or debug information to be present in binaries.
- ∗ It utilizes a compiler's intermediate representation (LLVM) in its internals which opens the domain of running existing source-level analysis and optimization passes built up over decades by hundreds of developers.
- ∗ It is evaluated and shown to recover accurate and precise information from C, C++, and Fortran binaries obtained from the SPEC2006 benchmarks suite; compiled using two different compilers in a reasonable amount of time.

2. Analysis and Rewriting Framework

Figure 1 presents an overview of SecondWrite [6], [3]; our executables analysis and rewriting framework. SecondWrite translates the input x86 binary code to the intermediate format of the LLVM compiler [2]. The disassembler along with the binary reader translates every x86 instruction to an equivalent LLVM instruction.

A key challenge in binary frameworks is discovering which portions of the code section in an input executable are definitely code. Smithson et. al. [20] 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 were 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 binary segments are scanned for values that lie within the range of the code segment. The resulting values are guaranteed to contain, at a minimum, all of the indirect CTI targets. More information on how we form functions and function boundaries using binary characterization is found in [11].

Memory stack analysis is done for every procedure to detect its corresponding memory arguments as explained in [3]. The techniques presented in [3] along with [4] are used to split the physical stack into individual abstract stack frames. Global and stack regions appear as arrays of bytes in the IR.

3. Decoding the floating point variables

In this section, we describe our technique to decode all the floating point stack operations and represent them in higher level code using floating point variables, function arguments and function returns, instead of the low level stack layout used in the assembly.

We begin by introducing the x86 floating point stack. The floating point hardware stack has a maximum height of 8 which means there are only 8 physical floating point registers that can be used at any time. The names of those registers, as used by the hardware instructions, are dynamic and are relative to the current top of the floating point stack. If we assume the fixed physical register names are: PST_0 - PST_7, then the assembly instructions will refer to another set of names ST_0 - ST_7, where ST_0 always refers to the register at the top of the stack. For example, if the height of the stack is one, then ST_0 refers to PST_0. If the stack is full, then ST_0 refers to PST_7. In general, ST_x is mapped to PST_y where $y = \textit{TOP}(\mathrm{I}) - 1 - \mathrm{x}$ where $\textit{TOP}(\mathrm{I})$ is the stack height at instruction I and $0 \le y < \textit{TOP}(\mathrm{I})$. Whenever a function returns a floating point value in a register, it pushes the value on the floating point stack. Whenever a function takes floating point values as arguments in registers, the caller pushes the values on the floating point stack. It is assumed that $\textit{TOP}(\mathrm{I})$ cannot be negative at any instruction I.

Decoding the floating point stack means mapping every assembly operand among ST_0 - ST_7 into a corresponding IR register among PST_0 - PST_7. To do so in the IR, we declare the registers PST_0 - PST_7 as local variables inside each procedure. It turns out from the previous equations that we only need to identify for every instruction I, what is the corresponding $\textit{TOP}(\mathrm{I})$ in order to decode the floating point operands successfully. This task is not trivial because of the existence of indirect and external calls.

If there is no indirect or unknown external call in the program, the problem is trivial because we can traverse the control flow of the program, tracking the floating point stack height at every point, and set the value of $\textit{TOP}(\mathrm{I})$ at every instruction I depending on the floating point operations observed. This analysis will not work in the presence of indirect and external calls because when we hit such a call, we will not know what function is being called and how the height of the stack will be affected by this call.

We use a symbolic analysis scheme to solve this problem by maintaining a symbolic value X_i for every indirect and external call i representing the difference of the floating point stack height before and after the call. Sometimes we refer to that difference as *StackDiff* in the paper. After doing the symbolic analysis, each $\textit{TOP}(\mathrm{I})$ will become a symbolic expression in terms of the X_is. We build symbolic linear equations to solve for X_is. Once the X_is are calculated, $\textit{TOP}(\mathrm{I})$ will be known for every instruction.

It is statically indeterminable to be able to decode the floating point operations correctly in all cases in the presence of indirect and external calls. In this work, we show that if we lay out some assumptions, we can actually guarantee a correct and functional representation of the floating point stack operations in all cases that adhere to those assumptions. Our assumptions are:

1. At control-flow join points, the floating point stack height must be the same for every predecessor basic block.
2. At indirect and external calls, the floating point stack height must be zero before the call.
3. Every indirect or external call can return at most a single floating point value on the floating point stack.

The above assumptions are correct in compiled code in every case in every compiler we are aware of. They are also true in most hand written assembly code, but may not be always true in theory. The justifications for the assumptions are as follows. (1) If the stack height is not balanced at join points, any subsequent floating point stack access will be indeterminable as it might access different values depending on the path taken at run time. (2) For indirect and external calls, the behavior of their targets is usually unknown to the compiler, and hence the compiler must assume they might use all the floating point stack registers, hence it has to clean the stack before such calls. We can state this assumption by saying we assume floating point registers are scratch registers. Theoretically, a compiler might know in some cases the behavior of the functions being called and may not clean the floating point stack, but practically we are not aware of such a compiler. (3) The last assumption above is coming from the fact that we are not aware of any calling convention that allows the return of more than one floating point stack register from indirect calls and externals.

We translate the above assumptions into the symbolic analysis propagation rules present in figure 2, explained as follows. For internal function calls, we use helper variables $Y(F)$ to represent the symbolic expression representing *StackDiff* of every function F. The executable is traversed in a depth first search manner starting from the entry point function for the binary, and from functions that are never called directly in the code. Then we analyze the remaining strongly connected components in the call graph. The assumptions (1) through (3) above represent the symbolic equations in lines (1) through (3) in figure 2. The actual values of X_is can only be zero or one because before the call, the stack height is zero according to assumption (2), and the call can return at most one value according to assumption (3). The height of the stack cannot go negative and hence the actual value of the X_is cannot be negative.

The symbolic equations represented by equations (1) through (3) in figure 2 along with the symbolic unknowns X_is are transformed into a linear system of equations. To solve those equations, we employ our custom linear solver that categorize the equations into disjoint groups based on the variables used in every equation, and then solve every group only if the number of equations is equal to the number of unknowns. We keep propagating calculated values to other groups until no more calculated values are present. Most of the X_is are usually solved using equation (3) in figure 2.

The remaining unknowns are assumed to take a value of $X_i = 1$ conservatively. This will be always correct because from our second assumption above, the stack height is zero before every indirect and external call. In this case, if we declare by mistake that a particular call modifies the stack height by adding one element; this element will never be accessed. In this case, even if there are subsequent floating point stack operations, they have to push values on the stack before reading them.

The floating point register arguments and returns are declared in the IR as follows: a) Whenever a function has *TOP*$(\mathrm{I}) > 0$ at its entry point instruction I, the function is declared in the IR to take as many floating point values as the value of *TOP*(I). They will be passed as arguments and copied to the correct local variables according to the mapping we described earlier. b) Whenever X_i or $Y(F)$ are greater than zero at a call site, this call site will be returning one or more floats in the IR and they will be copied to the corresponding local variables in the callers according to the *TOP*(I) value at the call site.

Unknown Symbolic Values :

X_i, where X_i = *StackDiff* of indirect/external callsite i

Helper Variables :

$Y(F)$ = *StackDiff* of function F, where F is an internal function
TOP(I) = top of the stack after executing instruction I
I' = the previous instruction to I. At a basic block (*BB*) entry, it is the first instruction of *BB*.

Initial Conditions :

Root functions "not called directly anywhere" as well as the entry point function have entry *TOP*$(\mathrm{I}) = 0$ where I is a NOP instruction inserted at the entry point of each of those functions.

Data flow rules :

```
At every basic block ($BB$) entry:
    $TOP(\mathrm{I}) = TOP(\mathrm{I_n})$, where $I_n$ are the terminators of the predecessors of BB ————(1)
For every instruction $I$:
    $I$ = push ... $\Rightarrow TOP(\mathrm{I}) = TOP(\mathrm{I'}) + 1$
    $I$ = pop ... $\Rightarrow$
        if ($TOP(\mathrm{I'}) = \mathrm{X_i}$) $X_i = 1$ ————(3)
        $TOP(\mathrm{I}) = TOP(\mathrm{I'}) - 1$
    $I$ = call $F \Rightarrow$
        if ($F$ is an external or indirect)
            $TOP(\mathrm{I'}) = \mathrm{zero}$ ———— (2)
            $TOP(\mathrm{I}) = \mathrm{X_i}$
        else
            $TOP(\mathrm{A}) = TOP(\mathrm{I'})$ where $A$ is the first NOP instruction in $F$
            Analyze $F$ to get $Y(F) = func(X_1, ..., X_n)$
            $TOP(\mathrm{I}) = TOP(\mathrm{I'}) + \mathrm{Y(F)}$
    $I$ = return from $F \Rightarrow$
        $Y(F) = TOP(\mathrm{I'}) - TOP(\mathrm{A})$, $A$ is the first NOP instruction in $F$
        $\forall Z$ = return from $F \Rightarrow TOP(\mathrm{Z}) = TOP(\mathrm{I'})$
```

Figure 2. Data flow rules used to decode the floating point stack

4. Function Prototypes Recovery

Detecting the complete and accurate set of function arguments and returns is essential in producing a high quality code that can run correctly if recompiled. If some arguments are missing, the code will not work correctly in all cases. If more unnecessary arguments are identified, the code will run correctly, but will be less understandable by users.

We show how to accurately identify the register arguments and returns. Existing techniques show how to identify the exact set of memory arguments. SecondWrite already uses a variant of the algorithm used by Balakrishnan et. al. [4] to identify memory arguments [3]. Surprisingly, we did not find any related work that recognizes correctly and accurately register arguments and returns. Not recognizing register arguments and returns is acceptable if the goal is to help human understanding of binaries (as for existing methods), but unacceptable if the goal is to generate correct rewritten code (as for our method.) Typical x86 codes have less register arguments than memory arguments, but they still have large numbers of register arguments especially for optimized executables.

A brute force algorithm for identifying register arguments and returns is to define the set of registers read without being initialized inside a procedure as arguments, and the registers modified inside a procedure and then later used at some of the call sites as returns. This technique will result in many spurious arguments since all registers which are saved and then restored back in a function (such as callee saves) will be declared as arguments and returns for this function, which is not true. Further, this algorithm might miss some arguments if not carefully implemented. For example, a procedure not accessing any register at all might be declared as taking no register arguments, which may not be true since it might be calling a function which is taking a register argument.

We propose below an algorithm which identifies accurately all register arguments and returns. Our algorithm is conservative since it will not miss any arguments. It is also accurate since it prunes out unnecessary extra arguments in many cases.

The main challenge in being accurate and yet conservative is that the stack locations used to save registers need to be tracked to make sure they are only used for this purpose, thus allowing those registers to be pruned from the arguments or returns. The stores of the register values at the beginning of the function should dominate the loads used to restore them back. There should not be any write to those stack locations in between. If those stack locations are read in the middle of a function, the corresponding registers must be declared as arguments.

Our register arguments and returns detection algorithm is composed of five steps. 1) We assume all registers are arguments to every function and there are no register returns. 2) We declare all registers written to inside a function or any of its callees as potential return registers. 3) We run our algorithm for detecting saved locations by detecting the set of stores to the memory stack which are never loaded back except before the return from the function. We call those store instructions *DeadStores* since they will be eventually removed from the code. For each of the detected dead stores, we determine the corresponding saved register and remove it from the potential returns set. 4) We run our algorithm to propagate the register arguments correctly and prune unused ones. 5) We prune the unused return registers out. Next, we describe each of those steps in details. Step 1 is trivial. We proceed from step two.

The second step in our algorithm is to detect the initial set of potential return registers. The simple idea is that any register which is being written to inside a function is a potential return register from this function. For example, if a function `foo` is calling function `bar`, and `bar` is modifying `eax`, then `foo` and `bar` will be declared as potentially returning `eax` despite the fact that there is no write to `eax` inside of `foo`. We do a post-order depth-first search traversal of the call graph (which visits child nodes before their parents) and propagate the set of potential return registers upwards in the call graph by looking for the written-to registers. Whenever we find a call to a function, we add its potential returns to the caller function potential returns. We handle recursion using a work list mechanism such that whenever we detect a call to a function which has not been analyzed yet, we add the caller function back to the work list.

After detecting the potential returns, we add them to the IR in every return statement inside every function. If more than one register is returned, we return a structure containing all combined potential return registers.

The third step in our algorithm is to detect the callee saves registers and exclude them from the list of potential returns. Since callee-saves values are saved to the memory stack, we need a memory analysis technique to track the memory stack locations where they are saved. Tracking memory in executables is not a trivial task. Our saved registers detection does not need a sophisticated memory tracking algorithm because it only needs to track stack memory. Neither heap nor global memory need to be tracked.

We modify the Value Set Analysis (VSA) algorithm proposed by Balakrishnan et. al. [4] by removing global and heap memory tracking, keeping only stack memory tracking. We also remove the context sensitivity from the algorithm since it is not needed in this application. The resulting algorithm is less powerful for general memory tracking but is sufficient for this purpose.

As a quick summary of the VSA algorithm, it derives a conservative estimate of the set of addresses and integer values every memory location and register can contain at any program point. Every set of values is represented as a strided interval with a lower

Algorithm 1: The callee-saves detection algorithm

```
   Input: A copy of the LLVM IR for a binary
   Input: PotArgs : maps functions to their potential register arguments
   Input: PotRets : maps functions to their potential return registers
   Output: DeadStores : maps functions to the dead register stores
   Output: PotRets : The input map after pruning saved registers
1  foreach reg $\in$ PotArgs do
2      Create a dummy register dummy ; DummyRegs(reg) = dummy
3  end
4  ADDRS = $\phi$
5  foreach Function F do
6      foreach Instruction I in F do
7          if I = store reg, Ptr AND reg $\in$ PotArgs then
8              if ValueSet(Ptr) = {address} (Singleton) then
9                  ADDRS = ADDRS $\cup$ {(reg,address,I)}
10             end
11         end
12     end
13     foreach (reg,address, I) $\in$ ADDRS do
14         allocate a dummy pointer DummyPtr((reg, address)) at the
           beginning of F
15         store DummyRegs(reg) to DummyPtr((reg, address))
16     end
17     foreach Instruction I in F do
18         if I is UnsafeInstruction(address) where (reg,address,X) $\in$
           ADDRS then
19             insert a volatile load from DummyPtr((reg, address))
20         end
21         if I = store value, Ptr AND ValueSet(Ptr) $\supseteq$ {address}
           AND (reg,address,X) $\in$ ADDRS then
22             insert a store value to DummyPtr((reg, address))
23         end
24         if I = load Ptr AND ValueSet(Ptr) $\supseteq$ {address} AND
           (reg,address,X) $\in$ ADDRS then
25             insert I' = load DummyPtr((reg, address))
26             for every use of I insert a cloned use of I'
27         end
28     end
29     Run LLVM Memory to Register Promotion on All DummyPtr
30     Run LLVM Dead Code Elimination on F
31     foreach (reg,address, I) $\in$ ADDRS do
32         if DummyPtr(reg, address) is deleted AND
           DummyRegs(reg) has no uses OR only used in return
           instructions then
33             DeadStores(F) = DeadStores(F) $\cup$ {I}
34         end
35         if DummyRegs(reg) has no uses OR DummyRegs(reg) is
           used in all return instructions of F then
36             PotRets(F) = PotRets(F) - {reg}
37         end
38     end
39 end
```

and upper bounds; and a stride. In our modified implementation of VSA, we only keep track of the lower and upper bounds.

Before we run the saved registers detection algorithm, we convert the registers inside of each function into the SSA form. This is straight forward; indeed in our implementation LLVM already does that. Our algorithm works on a temporary copy of the IR.

Algorithm 1 detects the dead stores used to save registers and prunes those saved registers from the potential return register set. Lines 6 through 12 in the algorithm collect the addresses on the stack that are used to store register values. For each of those addresses, a simple memory liveness analysis is being conducted using standard memory-to-register promotion and dead code elimination compiler passes (both these passes are already available in LLVM). Lines 13 through 16 create a dummy memory location in the IR for each pair of address and register identified. We initially store a dummy value we create to each one of those memory locations. Lines 17 through 28 examine the uses of every address using VSA. At every possible read of an address, we insert a load from the dummy memory location we create. At every possible write to that address, we insert a store to that dummy memory location of the stored value. After that, we run the memory-to-register promotion compiler pass again on those memory locations. Finally, lines 31 through 38 determine the final set of dead stores. If the dummy memory location is promoted successfully to registers, and the only use of the dummy value is at the return then it is saved and can safely be removed from the potential return. The corresponding initial register stores are declared to be dead in this case. If the same previous conditions occur and also there are other uses of the dummy value, then the register is removed from the potential returns, but the initial store is not dead and is considered a real use of the register; i.e. the register becomes an argument.

The *UnsafeInstruction*(*address*) functions appearing in line 18 in the algorithm is responsible of deciding whether the instruction may have side effects which can potentially access that *address*. External calls where any stack address appears in the value sets of one of the arguments are considered unsafe as they may do arithmetic on those addresses and potentially read from or write to our *address*. An example of this behavior is `memcpy`, `strcpy` and other string manipulating functions from the standard C library. Some external functions are pre-identified safe and known not to do arithmetic on pointer arguments. For example, we parse format strings of `printf`, `scanf` and similar functions and in some cases we can prove those functions are safe.

After detecting the dead stores used to save registers and pruning the callee-saves from the potential returns, we proceed to step four which identifies the actual register arguments. We traverse the call graph of the executable in post-order depth-first search traversal, which ensures child nodes are visited before their parents. For each potential register argument inside a function, we declare it as an argument if and only if we see a "real" use of this register in the function. If a register is used in a store instruction among the dead stores identified by algorithm 1, the store is not considered a real use. Uses in calls are only considered "real" if the callee takes the register as an actual identified argument. A work list mechanism is maintained to handle the dependencies between functions. PHI nodes that link multiple SSA versions of the same register are not considered uses and are tracked. Returns are not considered real uses because if the return is the only use of a register, there is no need to pass it as an argument. Propagating the actual return registers (step 5 in our algorithm) is done in a similar way to the one above except that it works on functions in the forward call graph order and looks for uses of return values at call sites.

The correctness of our register arguments and returns algorithm is guaranteed for internal functions. The reason is that we start our algorithm initially by having all registers as arguments, and then remove those which are not really used. For returns, we start the algorithm by adding all registers that are written to inside of a function or one of its callees, we then remove the ones which are unused at call sites. The correctness in the presence of indirect calls, external calls and call backs is described below.

Our algorithm runs the same way on indirect calls and is correct. At every indirect call, SecondWrite inserts a call translator function that checks the value of the function pointer and calls the corresponding IR function accordingly. In this case, this call translator is treated the same way as any normal function in this algorithm under the assumption that the call translator will call all possible target functions.

Regarding external calls, they are treated correctly by our algorithm in all compiler generated code where the external function has a standard compiler calling convention; ex; *cdecl*, *fastcall*, *thiscall*, *stdcall* and others. Some external functions like standard C and C++ libraries are known to SecondWrite; hence our algorithm will know from the prototypes what registers are needed passing. For the unknown prototypes, we pass all registers that form the union of all the possible known calling conventions, and return all possible returns from the same union. This is not efficient, but will produce correct code under the above assumption. We insert assembly instructions before the external call to make sure we pass the correct register values from the IR to the corresponding physical register in hardware, and copy the physical returns into the correct IR registers after the call. Only if the external call has a non-standard compiler calling convention is when we might not be able to handle it correctly. We never experienced any such external call in all our tested programs.

5. Variable and Type Recovery

In this section, we present our techniques to recover source-level variable information from executables, and then present them with meaningful data types in the IR. Our techniques focus only on memory allocated variables. Register-allocated variables can be handled after detecting register arguments and returns using any compiler liveness analysis that detects a variable for every live range of a register in the executable.

Variable and type recovery from executables is a hard problem because symbol tables are absent. Every memory-allocated variable access in the source code is represented by a memory store or load in the executable. Those memory accesses are either direct accesses to locations represented by constant addresses, or indirect memory accesses to locations represented by some register value. Direct memory accesses can be used to infer variable information by examining the constant memory address being accessed, but indirect memory accesses are unknown accesses and need more advanced memory analysis to reveal the underlying memory locations. That is why pointer analysis is important while recovering variables and data types from executables since it reveals what are the possible memory locations an indirect memory reference can possibly access.

Researchers in this field know this and the best known variable identification technique from executables (DIVINE [5]) uses an advanced memory analysis technique called value set analysis [4], which is a generalized form of alias analysis. DIVINE presents accurate variable identification that detects 88% of the memory-allocated variables in executables. The problem with DIVINE is that it is not scalable and requires a very long time to analyze even small programs. Our aim is to present techniques with the same accuracy as DIVINE, but run orders of magnitudes faster.

Our key insight that enables scalability is that efficient variable detection and type recovery do not require a sound pointer analysis. Unsound pointer analysis usually means incomplete points-to sets. As an example, if variable `x` points to `y` and `z`, an unsound pointer analysis might report `x` points to `y` only. Usually unsound pointer analysis is unacceptable, but variable detection from executables is a best-effort analysis and no method claims to detect 100% of the variables. If we are going to miss some variables anyways because of the nature of the problem we are solving, then we can sacrifice the soundness of the analysis at the expense of losing some variable information – as losing variable `z` in the given example above, but with the gains of having a practical analysis that scales well for large executables.

The correctness of the recovered IR, while missing some variables due to the unsound pointer analysis, comes from the fact that the relative ordering between variables in the memory layout is

store y, x (store value y to location x of size S)	$\forall\, z \in$ ALocs(PtSet(x)) : PtSet(z) $\cup$ = PtSet(y) **Variables:** UpdateALocs (PtSet(x), S)
y = load x (load location x of size S to y)	$\forall$ z $\in$ ALocs(PtSet(x)) : PtSet(y) $\cup$ = PtSet(z) **Variables:** UpdateScalar (PtSet(x), S)
$y = x$	PtSet(y) = PtSet(x)
$y = x + z$, PtSet(x) is not empty	**if** z is a constant **then** PtSet(y) = PtSet(x) $>>$ z **Variables:** **if** z is a constant **then** UpdateStructure (PtSet(x), z) **else if** z has SCEV bounds and stride **then** UpdateArray (PtSet(y), stride, bounds)

Table 1. Points-to sets propagation and variable detection rules

A = call foo (arg_1, ..., arg_n)
foo has the known prototype:
$retType$ foo ($type_1$, ..., $type_n$)
	$\forall$ x $\in$ [1,n]
	setType(arg_x, $type_x$)
	setType(A, $retType$)

$A = B$ op C $op \in \{+, -, *, /, \%, >>, <<\}$ op has type: $opType$ A, B, C has empty points-to sets	setType($\{A, B, C\}$, $opType$)
A = load B store A, B	unifyType(A, ALocs(PtSet(B)))
op_1 = ϕ (op_2, ..., op_n) op_1 = typecast op_2 to $type$	unifyType($\{op_1$, ..., $op_n\}$)

Table 2. Typing rules

maintained in the recovered IR. For example, if we detect two integer local variables at offsets 0 and 20 on a stack frame of size 24 bytes, we will lay out those variables in a structure which has the following three members: a) An integer in the range [0-3]. b) A generic array of bytes in the range [4-19]. c) An integer in the range [20-23]. Preserving the layout of the variables in such a structure maintains the correctness of any indirect memory access to this region. The arrays inserted fill the unknown gaps between variables and maintain the memory layout. This representation helps understanding what variables are detected along with their types, and at the same time maintains the functionality of the rewritten program.

We introduce the concept of a best-effort pointer analysis; where the identified points-to set of each pointer may not be complete, but we terminate the analysis in a certain amount of time nevertheless to prevent it from taking too long even before it converges. This analysis is not correct given the usual criteria for correctness, but suffices in the way we use it to identify as many discrete variables as possible. Our best-effort pointer analysis is a flow and context insensitive data flow analysis that has the following properties:

* It limits the cardinality of the points-to sets to a fixed number.
* It does not track interprocedural information via indirect calls.
* The number of analysis iterations is set to a fixed number.

Having the above relaxations makes our analysis much faster at an extremely small loss in precision. The intuition behind this is as follows: a) A flow and context sensitive pointer analysis is not needed since the variables usually have the same size and type in all flows and contexts of a program. Some exceptions to this might happen which is not common in the programs. b) Limiting the cardinality of points-to sets does not affect the precision that much since only few variables will have large points-to sets. c) Propagating interprocedural information through indirect calls will only affect functions which are only called indirectly. Those functions are still analyzed, but their arguments will have unknown points-to sets. Given that there are relatively few such functions in executables, skipping their arguments propagation is not a big loss. d) Limiting the total number of iterations will only affect longer chains of pointers. For example, the first iteration will always reveal some pointers. The second will reveal two-level (double) pointers. Subsequent iterations reveal more pointer levels. Usually most variables do not have more than four level pointers, which means subsequent iterations will only reveal very little information.

5.1 Best Effort Static Variable Recovery

We show in this section how a simple best-effort pointer analysis can be used for identifying variables. This pointer analysis should be suitable to run on executables where no variables yet identified. We could have modified current memory analysis schemes on executables like [4] to fit our needs, but we show a simpler analysis with similar precision and much better scalability.

Before we begin the analysis, we identify all base memory regions in the executable. An executable has three base memory regions. 1) The global memory region where global variables are located. 2) The stack memory region where local variables inside functions are located. Stack regions are allocated at the beginning of a function and deallocated at the end of the function. SecondWrite already represents those as large arrays in every function. 3) The heap memory region where dynamically allocated variables are usually located. Those are identified by detecting calls to functions like `malloc` and `new` in the executable.

Every detected memory-allocated variable is represented by an abstraction called *ALoc* which stands for Abstract Location. The name is similar to the name used by DIVINE [5]. An *ALoc* contains an offset inside a base memory region and a size representing the variable size. Variables allocated to registers are represented by IR symbols which represent the SSA form of those registers.

Our pointer analysis conservatively assumes that every detected variable can be a pointer. We assign points-to sets to every IR symbol and detected ALoc. When the analysis is done, the actual pointers are identified by tracking if the corresponding points-to sets are not empty.

We implement the points-to sets using the efficient LLVM sparse bit vector data structure. For every base memory region, we assign it a series of unique bits where the number of bits equals the size of the region in bytes. If the size of the base memory region is not known (usually in heap allocated arrays), we assume an arbitrary size. This allows us to detect variables with offsets up to that size. Whenever an access is detected beyond that arbitrary size, we do not track it. This is an important part of our best-effort analysis that allows us to recover a subset of the variables on unsized base memory regions instead of totally giving up on them as the case in DIVINE [5]. Whenever a symbol or an ALoc points to some variable in a certain memory region, the bit corresponding to the starting address of the variable will be set to one. The number of bits set to one equals the number of variables pointed to by a symbol or an ALoc.

Table 1 shows our detailed propagation rules for the best-effort pointer analysis as well as for detecting the variables. We introduce the following definitions to ease the understanding. 1) PtSet(x): takes an ALoc or an IR symbol x and retrieves its points-to set 'bit-vector'. 2) ALocs(x): takes a bit-vector x and retrieves the set of ALocs starting at the addresses that correspond to the set-bits in the bit vector x. 3) UpdateALocs(x,y): takes a bit-vector x and a size y and creates ALocs starting at the addresses corresponding to the set-bits in the bit-vector x with the given size y. If existing ALocs overlap the new ALocs, the new and old ALocs will be split into smaller ALocs to avoid the overlap. 4) UpdateStructure(x,y):

takes a bit-vector x and a number y. It defines a set of structures starting at the addresses corresponding to the set-bits in the bit-vector x. Each structure has its last member at offset y. If a structure already starts at one of the starting addresses, its last member offset will be updated with the maximum of the existing offset and the new one (y). 5) UpdateArray(x,y,z): takes a bit-vector x, a number y representing a stride, and another number z representing the upper bound of the array. It defines arrays starting at the addresses corresponding to the set-bits in the bit-vector x. Each array has a maximum size z. The arrays will be declared to have an element size y. Existing arrays will be merged with the new declared ones and the element size will be set to one if overlapping arrays have conflicting element sizes.

Here we describe briefly the propagation rules in table 1. For a store instruction, the points-to sets of the ALocs pointed to by the pointer operand will be unioned with the points-to set of the value stored. This is called a weak update in the domain of pointer analysis. A load will set the loaded value points-to set to whatever is pointed to by the pointer operand. Stores and loads will create ALocs as they are resolved using the UpdateALocs function described earlier. For pointer arithmetic, the points-to sets will be shifted right according to the positive constant added. If the constant is negative, the shift will become to the left. Adding a constant to a pointer is a hint about the existence of a structure where the pointer address is the start address, and the constant represents one field offset inside the structure. We use this hint and declare a structure identified by the starting address and the last member offset. The structure's last member offset might be updated in subsequent pointer arithmetic operations that start from the same base. The structure's last member offset will eventually be the maximum observed constant that was added to the pointer in the program. Adding a non-constant value is an indication that an array exists. An array will be declared in this case. We use the Scalar EVolution (SCEV) analysis by LLVM to deduce the bounds and the stride of the arithmetic and use this information to describe the array. If such information is not present, we do not declare an array.

The more pointer analysis rounds done, the more ALocs, structures and arrays are identified in all base memory regions. More pointer analysis rounds help identifying multi-level pointers since the first round will always reveal single level pointers. The second round will propagate the points-to sets for those ALocs and identify their points-to sets leading to the identification of two level pointers. More rounds will reveal more levels.

After all iterations are done, collected information about arrays gets resolved. For every base memory region, we fill in the gaps between ALocs using arrays. The bounds and stride information are available from our earlier propagation. If no bounds are available, previously defined ALocs are used as bounds. If no stride information is available, a stride of one is used which means the array is an array of bytes. Overlapping arrays are combined into one bigger array as described earlier.

At the end of this process, a structure hierarchy is created based on the structure information calculated for every base memory region. Using the starting and ending offsets previously calculated for every structure, we construct nested hierarchy structures. We define inner and outer structures such that any outer structure must have its starting address less than any starting address of any nested inner structure, and its ending address larger than any ending address of any nested inner structure. A straight forward algorithm is employed to produce this hierarchy.

5.2 Data Type Recovery

Data type recovery aims at representing every symbol in the IR with a meaningful type. It declares a map between every symbol in the IR and the corresponding detected data type. It uses this map to rewrite the complete IR such that the instructions use the detected types instead of the generic types that are used by SecondWrite.

Without integrating type recovery with some pointer analysis, detected types will be less accurate because of two reasons: 1) Instructions like memory loads and stores will usually be untyped since there is no memory tracking possible. 2) Multi-level pointer types will not be detected because there is no way to track them without having some sort of pointer analysis.

To achieve the goal of typing memory accesses and IR symbols; and detecting multi-level pointer types, we integrate our best-effort pointer analysis and variable recovery techniques described above with our type recovery system. Any other pointer analysis like [4] can be theoretically used, but will be orders of magnitude slower which makes it less practical in large executables. That is the disadvantage of TIE [14] which is the state of the art binary type recovery technique.

Integrating our variable identification system with type recovery makes the type recovery simpler because it will need only recover scalar types like integers, floats and doubles. Structures and arrays are detected as part of the variable identification. A pointer is detected if the points-to set of the corresponding ALoc or IR symbol is not empty. In this case, we get the ALocs pointed to by that pointer and type them according to our rules. We keep doing this for longer pointer chains as needed.

Table 5.1 shows the most important typing rules we have. There are two main type sources. a) Known external function calls like standard C/C++ library calls. For those, we set the types of actual arguments passed to be the same as the known argument types from the prototypes and we do the same thing for the return value. b) Arithmetic operations with non-pointers: in this case the type is deduced from the semantics of the operation itself – whether it is an integer or a floating point operation –. We use the function *setType* to update the type of the symbol or the ALoc in the type map we declare. For pointer types, we type the ALocs represented by the points-to sets of the corresponding variables.

For the other operations in the table, we propagate the types using the function *unifyType*. This function attempts to set the data type of all the given symbols and ALocs to be the same. At least one of the symbols or the ALocs given to that function should be typed. Whenever this function finds conflicting types, it gives up and does not update any types. It is used for copy operations like type casts and phi nodes. It is also used to propagate types through memory as shown in the rules for stores and loads. Interprocedural information is propagated by unifying the formal and actual arguments types at a call instruction. The return value data type at the call site is unified with all the data types of all return values appearing in the return statements inside the called function body.

IR Correctness. We are able to produce a correct and functional IR even if we do not detect some variables and data types. To be able to do that, we rewrite the IR using the following restrictions:

1. We use generic types for the symbols we could not detect types for. The generic types will be wide enough to handle the largest possible variable size that can be allocated to a physical register in the hardware. Type casts are used as needed to convert the generic type to actual types used in different operations.
2. We never assign a type to an IR symbol that conflicts with its use. For example, if we see a 4 byte load, we will never type the pointer as a pointer to short (2 bytes) even if our analysis detects it this way. Otherwise, the load will be wrong.
3. All variables identified for a certain memory allocation will be surrounded by a structure data type. The order of the variables inside that structure is the same as the order they appear in the original executable. The memory regions with no variables declared will be declared as arrays of bytes and will be placed at

Application	Lang	# Inst	# Proc	Time(s)
mcf	C	3,357	36	0.15
lbm	C	7,740	30	0.11
astar	C++	12,677	111	0.39
libquantum	C	13,800	73	0.41
bwaves	F	19,002	22	0.87
bzip2	C	21,408	51	1.14
sjeng	C	32,238	121	2.86
milc	C	34,183	172	2.38
sphinx	C	41,669	210	6.68
leslie3d	F	43,432	32	2.78
hmmer	C	85,981	242	5.29
namd	C++	103,365	193	11.71
soplex	C++	116,743	1523	20.09
zeusmp	F	118,429	68	5.44
omnetpp	C++	148,453	3980	59.58
h264	C	170,684	462	19.78
gobmk	C	196,230	4188	35.34
cactus	C	218,896	962	25.57
povray	C++	288,957	3678	72.49
perlbench	C	313,036	2183	67.89
gromacs	C/F	396,450	674	38.14
calculix	C/F	506,725	771	54.79
dealII	C++	766,555	15619	815.05
gcc	C	934,292	6426	354.68
tonto	F	1,303,359	2878	342.99

Figure 3. Benchmarks Table

the correct offsets inside those structures. This guarantees that every unresolved pointer arithmetic will still point to the correct variable in the rewritten executable.

6. Results

In this section, we present the results showing the effectiveness of our schemes to identify variables and data types. We first show results on the overall variable and data type detection process and then we show specific in-depth results for floating point variables and function prototypes. We evaluate our techniques on the SPEC2006 benchmark suite which represents C, C++ and Fortran executables using different optimization levels and compiled using two different compilers (GCC 4.3 for Linux, and Visual Studio 2010 for Windows). We use a machine with an Intel Core i7 3.33GHz processor with 24 GB of RAM.

All the recovered code in all the experiments was recompiled using LLVM 3.0, linked using GCC (Linux) and MinGW (Windows), and then tested on the ref and test inputs provided by the SPEC2006 test suite. All rewritten executables worked successfully and produced the correct answer as provided in the test suite. In the following sections, we show our detailed analysis results.

6.1 Variable and data types detection

In this section, we show the accuracy, scalability and quality of the recovered variables and types and compare them to the state of the art. We compile C benchmarks from SPEC2006 with all debugging information present and only use them for comparison. We currently do not support reading complete debugging information for C++ and Fortran, yet we collected results on those benchmarks without comparing with source code.

The first experiment shows the quality of the recovered variables using the same metrics DIVINE [5] used for comparison purposes. DIVINE [5] compares recovered variables in the binary to corresponding variables in the source code of those binaries to determine how well it did. It defines four variable categories as a result: 1) a matched variable is a recovered variable whose exact size and position matches the variable from the source code. 2) An over refined variable is when the source code variable is divided into more recovered variables; for example, an integer identified as four characters. 3) Under refined variables which are recovered as part of a larger source code variable ; for example, an un-identified structure member. 4) An unknown variable is a variable which is not one of those mentioned categories.

As shown from figure 4, an average of 86% of the variables are matched to the debugging information. We run this experiment on programs ranging from 2,149 instructions (mcf) to 934,292 instructions (gcc). DIVINE [5] reports an average of 88% matched variables on programs ranging between 252 to 5,371 instructions. This shows that our schemes has comparable precision to DIVINE [5] but on much bigger benchmarks. The largest benchmark they report variables results on is *deltablue* with 5,371 instructions.

The scalability of the variables and type detection is shown in figure 7. Our analysis scales linearly with program size for larger binaries. The detailed benchmarks sizes and analysis time are shown in table 3. The analysis takes around 6 minutes to analyze *tonto* which is a Fortran benchmark whose size is 1.3 million instructions. The average analysis speed is 1.7 seconds per 10000 instructions compared to 10 minutes per 10000 instructions in DIVINE. Thus our method is 352X faster than DIVINE on average. As mentioned before, the underlying reason for our much-faster analysis is using an underlying best-effort pointer analysis that is not guaranteed to have complete points-to sets. We consider that while recovering the IR to maintain correctness as we discussed earlier in section 5. dealII is the only program (out of 25) that did not scale well. dealII has very large number of procedures as shown in table 3. The interprocedural data flow propagation took most of the time in dealII. Still, it is finishing in around 13 minutes given that it has 766,555 instructions.

In order to evaluate our type analysis techniques, we calculate the same metrics that TIE [14] uses. TIE defines a type range for every variable recovered from the executable. An ordering between basic types is specified by a type lattice shown in their paper. The first metric they define is the *distance* which is the difference between the lattice heights of the upper and lower bounding types for each type range. The smaller the *distance*, the more accurate the identified types are. The maximum distance is 4. They also define their detected type range to be *conservative* if the actual source code type falls inside the detected range.

In order to compare with TIE [14], we define a range of types for every variable we detect where the lower bound is the single detected type by our analysis and the upper bound is the generic `reg32_t` type they define in their lattice. Based on that range, we calculate our distances and conservativeness rates.

In addition to the distance and conservativeness, we define our own metric that measures the precision of multi-level pointers detection. TIE metrics do not show how multi-level pointers are precisely typed since all pointer types have the same height on their lattice [14]. Our precision metric is defined as the ratio between the correctly recovered pointer levels to the source level pointer levels. For example, if a variable has a double pointer to integer type (`int**`) in the source code and we identified it as a single pointer to an integer (`int*`), then we identified one level only out of the three levels in source, which are *pointer* to *pointer* to *integer*. Our precision in this example will be 33%.

Figure 5 shows the conservativeness as well as the precision of our detected types. The conservativeness rate is 96% on average which is slightly higher than 90% that TIE reports. Our precision metric shows that we detect 73% of the pointer levels on average. The average distance detected for our type recovery system is 1.7 which is slightly better than the distance of 2 that TIE [14] reports.

Some of the larger binaries have lower type precision than other smaller ones. This is expected since larger programs tend to have more higher level pointers than smaller ones and those are usually

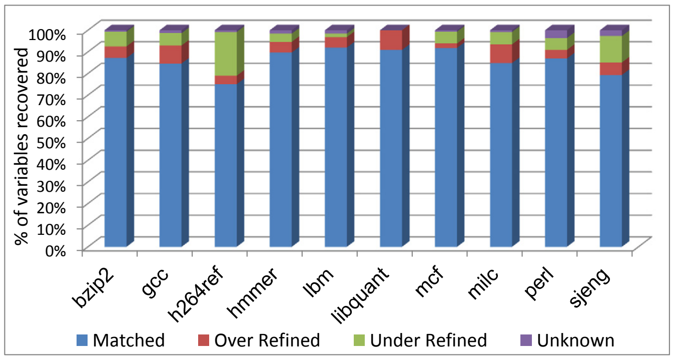

Figure 4. Accuracy of variable detection

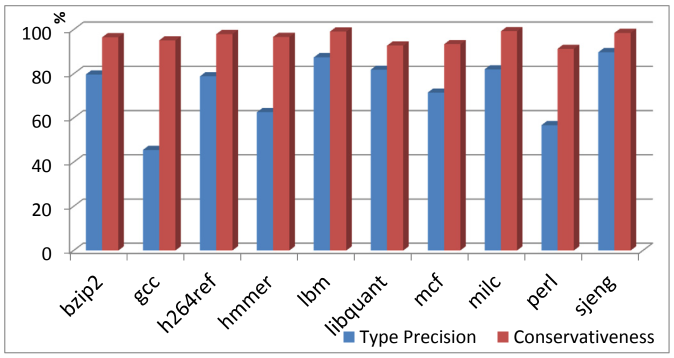

Figure 5. Accuracy of type detection

hard to detect since they rely on the effectiveness of the underlying pointer analysis. The conservativeness and distance measures used by TIE do not capture this fact as it is clear from figure 5.

It is worth mentioning that our variable and type recovery are integrated together in our system. The scalability shown in figure 7 as well as the detailed analysis time results shown in table 3 are capturing both the variable recovery and the type analysis.

6.2 Decoding the floating point stack

In this section, we show the effectiveness of our techniques in identifying floating point stack variables. We show the percentage of the symbolic values that were not solved using our linear solver and required the conservative assumption of $X_i = 1$. As mentioned in section 3, the main challenge while decoding the floating point stack is to identify whether an indirect or an external call is modifying the floating point stack height. According to our assumptions, whenever we are not sure about an indirect or an external call site, we decide conservatively that it is modifying the floating point stack by pushing a single value. We show how often we took that conservative decision in different binaries.

All register allocated floating point stack variables were recovered correctly and all the rewritten benchmarks ran correctly and produced correct answers. The conservative decision taken does not affect correctness as we explained in section 3. It only adds extra return values to some indirect and external calls and this might reflect adding more return values to internal functions as well. The next results section quantifies this effect.

On average, we took the conservative decision 28% of the time for non-optimized executables and 25% of the time for optimized ones. This means we are able to identify the exact floating point arguments and returns for more than 72% of the indirect and external calls on average. We are not aware of any work that identifies such information. Optimized binaries often have less variables than non-optimized binaries which translates to less floating point stack usage and less number of times when the conservative decision is taken. The conservative decision is usually taken more often in C++ binaries because they have more indirect calls with more straight line code and smaller functions than C and Fortran binaries, which translates into smaller number of equations.

6.3 Register Arguments and Returns

In this section we show the accuracy of the detected register arguments and returns. We run our algorithm only for the C and C++ benchmarks shown in table 3 and present the average number of added register arguments and returns (false positives). We never had any false negatives in any of the binaries we tested. We could not compare Fortran binaries since currently, we do not support reading Fortran prototypes from debugging information.

As shown from the figure 6, the average number of false positive arguments is 0.2 per function. The average number of false positive returns is 0.44 registers per function. These results include the conservative floating point returns we declare in our analysis, which explains why the average number of returns is higher. C++ executables tend to have more indirect calls than C executables which explains why they have more false positives.

In contrast to the work in [7], our method has three advantages: (i) it is guaranteed to discover all arguments; (ii) it has been demonstrated on a much larger programs; and (iii) it is orders of magnitude faster. First, their method cannot guarantee full coverage of arguments and returns because of being a dynamic analysis. Any unused argument or return during an execution trace can be missed. Missing arguments or returns is acceptable for human understanding of binaries, but unacceptable for rewriting binaries. Second, although our method produces slightly more false positives then their method (0.2 vs. 0.15 false positive arguments per function), it has been evaluated on far more functions (48,854 functions for our method, vs. just 13 functions for theirs.) Third, our analysis is much faster: for example, it takes only 30 seconds to analyze a program like *soplex* which has 116,743 instructions containing 1,523 procedures and produces prototypes for all of them. In their case, they need the same 30 seconds to only extract `MD5_Final` which is a single function of 67 instructions. This shows that our analysis is two to three orders of magnitude faster than their method, at the expense of a small loss in precision.

7. Related Work

Throughout the paper, we compared our work with the most recent work done in the areas of variable and type recovery [5, 14] and function prototypes identification [7]. In this section, we discuss other work that is relevant to our techniques.

Binary rewriting has been considered by a number of researchers. There are two main categories when talking about binary rewriters, dynamic binary rewriters and static binary rewriters. Dynamic binary rewriters rewrite the binary during its execution. Examples are PIN [16], BIRD [18] and others. None of the dynamic binary rewriters found produce high-level compiler IR. Examples of existing static binary rewriters include ATOM [13], PLTO [19] and UQBT [8]. None of those binary rewriters employ a compiler level intermediate format, like LLVM IR or similar; rather they define their own low-level custom intermediate format. They do not detect high level features such as floating point stack variables, register arguments to functions and data types.

Boomerang [12] is an open source decompiler. It has very limited capabilities and cannot handle large binaries. Register arguments has to be specified manually. It does not detect any floating point stack operations. Zhang et al. present a technique to re-

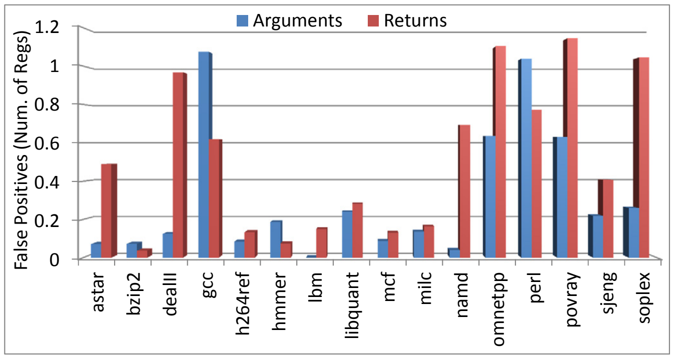

Figure 6. Accuracy of register arguments and returns

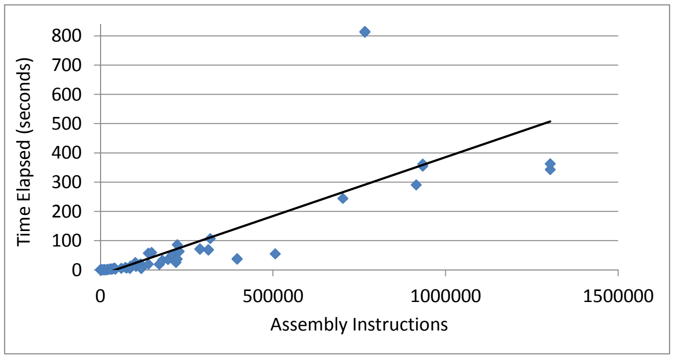

Figure 7. Scalability of variable and type detection

cover function arguments and returns from executable [22]. Their technique is similar the brute force technique described in section 4 which leads to imprecise results. Another technique recovering function prototypes is presented in [9]. It defines a language that can be used to specify machine independent calling conventions. It depends on ABI standards to recover the calling conventions.

REWARDS [15] presents a dynamic type recovery technique; TIE [14] shows better precision than REWARDS. We already compared to TIE [14] in our results. A technique to automatically reconstruct data types from binaries is presented in [10]. It is used in a tool that aims to produce C code from binaries; however no actual C code generation is demonstrated. One main disadvantage in their work is they do not track memory. As we have shown, tracking memory is very important in identifying accurate types. The analysis they produce is intraprocedural which limits its accuracy. Their algorithm is used by Torshina et. al. [21] in another attempt to reverse engineer data types in a tool named TyDec for program decompilation. An early work on type construction from binaries is by Mycroft [17]. It tries to construct C code from binaries with correct type information. However, it does not actually show results producing C code. The algorithm does not track memory locations and it is not clear if it can produce valid IR or C output code.

We are not aware of any work done to recover floating point stack variables except Hex-Rays [1]. Hex-Rays produces inline assembly in case it cannot resolve the variables which is not acceptable for our goal. As far as we know, their work is not published.

8. Conclusion

This paper shows how an executable can be represented by a compiler IR with source code level variables, data types and function prototypes. The analysis we present in this paper is scalable to large executables which makes it more practical than current techniques. The obtained high level IR is guaranteed to work correctly for compiled executables. The schemes are shown to work on executables containing up to million instructions.

References

[1] Idapro, Hexrays. `http://www.hex-rays.com/idapro/`.

[2] The LLVM Compiler Infrastructure. URL `http://www.llvm.org`.

[3] K. Anand, M. Smithson, K. ElWazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *EuroSys*, 2013.

[4] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.

[5] G. Balakrishnan and T. Reps. DIVINE: Discovering variables in executables. In *VMCAI*, 2007.

[6] R. Barua and M. Smithson. Binary rewriting without relocation information, May 24 2010. US Patent App. 12/785,923.

[7] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *NDSS*, 2010.

[8] C. Cifuentes and M. V. Emmerik. UQBT: Adaptable Binary Translation at Low Cost. *Computer*, 33(3):60–66, Mar. 2000.

[9] C. Cifuentes and D. Simon. Procedure abstraction recovery from binary code. In *Software Maintenance and Reengineering*, 2000.

[10] E. Dolgova et al. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software*, 35: 105–119, 2009.

[11] K. Elwazeer, K. Anand, M. Smithson, A. Kotha, and R. Barua. Recovering function boundaries from executables. Technical report, 2013. URL `http://www.ece.umd.edu/~barua/function-boundaries.pdf`.

[12] M. Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *Working Conference on Reverse Engineering*, 2004.

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

[14] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. In *NDSS*, 2011.

[15] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *NDSS*, 2010.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40:190–200, June 2005.

[17] A. Mycroft. Type-based decompilation. In *Proceedings of the 8th European Symposium on Programming*, 1999.

[18] S. Nanda, W. Li, L.-C. Lam, and T.-c. Chiueh. BIRD: Binary Interpretation using Runtime Disassembly. In *CGO*, 2006.

[19] B. Schwarz et al. PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture. In *In Proc. 2001 Workshop on Binary Translation*, 2001.

[20] M. Smithson, K. Anand, A. Kotha, K. Elwazeer, N. Giles, and R. Barua. Binary rewriting without relocation information. Technical report, 2010. URL `http://www.ece.umd.edu/~barua/without-relocation-technical-report10.pdf`.

[21] K. Troshina, Y. Derevenets, and A. Chernov. Reconstruction of composite types for decompilation. In *Source Code Analysis and Manipulation (SCAM)*, 2010.

[22] J. Zhang, R. Zhao, and J. Pang. Parameter and return-value analysis of binary executables. In *COMPSAC*, 2007.