Scalable Variable and Data Type Detection in a Binary Rewriter

Abstract

We present scalable static analyses to recover variables, data types, and function prototypes from x86 executables and obtain functional intermediate representation (IR) for analysis and rewriting purposes. Our techniques can run 352X faster than current techniques and still produce 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.

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 help, with the latter producing (non-executable) C-like pseudocode text. However, existing reverse engineering tools do not exhibit various 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 or recover precise information at the cost of scalability. For example, DIVINE, the most precise variable identification tool proposed in the literature, spends two hours while analyzing programs of the order of 50000 assembly instructions.

Recovering a functional intermediate representation (IR) in a scalable and accurate manner would be invaluable to security professionals. It would enable them to write compiler passes to analyze the IR to extract properties of interest. It would also allow the recovered IR to 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 code for which the source code has been lost. With a tool to recover IR, users of legacy code can fix bugs in such codes, modify the IR functionality or even port the code to new hardware systems. They may also optimize old code by doing more aggressive optimizations or making portions of the code parallel.

In this work, we present static analysis techniques that can recover source level variable and type information from x86 binaries as large as million instructions in 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 intermediate representation (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.

Unlike current binary analysis techniques, our recovery mechanisms are able to handle variables allocated on the floating point stack. Recovering such variables is a hard problem in the presence of unknown indirect and external calls. As we explain in later sections, the task of recovering these variables is imperative for obtaining a function IR from executables.

Our techniques will be able to accurately identify all register allocated variables that are used as arguments and returns from functions. Most x86 binary analyses techniques available today will only handle memory allocated arguments, but they do not present any methods for precisely handling register allocated ones. This is acceptable when recovering pseudocode, but unacceptable when recovering functional code. Little work is available that is doing either brute force techniques that are imprecise, or using dynamic analysis to detect arguments and returns which is precise but might miss some arguments and returns depending on the fired execution traces.

We enable variables and type recovery analyses that can handle executables as large as a million instructions in minutes and still produce precise source level information. Current state of the art analyses that produce the same kind of information are not scalable to such large programs. The inherent reason is that they have hard assumptions about the soundness of their analyses that we show can be relaxed and still give the same precision.

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.
* It is scalable to large programs since it is orders of magnitude faster than current analysis techniques that produce similar information from executables.[7][17].
* It presents algorithms for solving problems missed while analyzing executables like resolving floating point stack accesses and accurately identifying interprocedural interfaces.
* 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 binaries of all C, C++, and Fortran benchmarks in SPEC2006 compiled using two different compilers in a reasonable amount of time.

We recognize that rewriting all programs statically is a very challenging problem. This work should be seen as what it is: building on the first successful attempt to statically recover a functional compiler IR from executables, rather than the last word. We do not claim that we have fully solved all the issues involved; statically handling every program in the world may still be an elusive goal. Some malwares are packed; unpacking tools such as [8] may need to be run. Currently, a very small fraction are obfuscated or self-modifying; we expect future work incorporating dynamic feedback to the static rewriter will be helpful in such cases. However, the
resulting experience of expanding the static envelope as far as possible is a hugely valuable contribution to the community.

2. Analysis and Rewriting Framework

Figure 1 presents an overview of SecondWrite; the executables analysis and rewriting framework we use. 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. [8] proposed speculative disassembly, coupled with binary characterization, to efficiently address this problem. SecondWrite speculatively disassembles the unknown portions of the code segments as if they are code. However, it also retains the unchanged code segments in the IR to guarantee the correctness of data references in case the disassembled region was actually data.

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

The indirect CTIs are handled by appropriately translating the original target to the corresponding location in IR through a runtime translator. Each recognized procedure (through speculative disassembly) is initially considered a possible target of the translator, which is pruned further using alias analysis.

Above method is not sufficient for discovering indirect branch targets whose addresses are calculated in the binary. Hence, various procedure boundary determination techniques, like ending the boundary at beginning of the next procedure, are also used [5] to limit the possible targets.

The procedures discovered by the disassembler have all the physical registers as initial arguments. Memory stack analysis is done for every procedure to detect its corresponding memory arguments. The techniques presented in [5] along with [6] 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 $ST_i$ always refers to the register at the top of the stack. For example, if the height of the stack is zero, 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 $y = TOP(I) - 1 - x$ where $TOP(I)$ is the stack height at instruction $I$ and $0 \leq y < TOP(I)$. 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 in registers, the caller pushes the values on the stack. It is assumed that $TOP(I)$ 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 $TOP(I)$ 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 $TOP(I)$ 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. We traverse the control flow graph forwards while propagating the value of $TOP(I)$ 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. We traverse the control flow graph forwards while propagating the value of $TOP(I)$ 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. We traverse the control flow graph forwards while propagating the value of $TOP(I)$ 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. We traverse the control flow graph forwards while propagating the value of $TOP(I)$ 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. We traverse the control flow graph forwards while propagating the value of $TOP(I)$ 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. We traverse the control flow graph forwards while propagating the value of $TOP(I)$ 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. We traverse the control flow graph forwards while propagating the value of $TOP(I)$ 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. We traverse the control flow graph forwards while propagating the value of $TOP(I)$ 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.
Unknown Symbolic Values:

\[ X_i, \text{ where } X_i = \text{StackDiff of indirect/external call site } i \]

Helper Variables:

\[ Y(F) = \text{StackDiff of function } F, \text{ where } F \text{ is an internal function} \]
\[ TOP(1) = \text{top of the stack after executing instruction } I' \]
\[ I' = \text{the previous instruction to } I. \text{ At a basic block (BB) entry, it is the first instruction of BB}. \]

Initial Conditions:

Root functions “not called directly anywhere” as well as the entry point function have entry \( TOP(1) = 0 \) 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:

\[ TOP(1) = TOP(I_a), \text{ where } I_a \text{ are the terminators of the predecessors of BB} \]

For every instruction \( I \):

\[ I = \text{push } \Rightarrow TOP(1) = TOP(I') + 1 \]
\[ I = \text{pop } \Rightarrow \]
\[ \text{if } (TOP(I') = X_i) \quad X_i = 1 \quad \text{----------(3)} \]
\[ TOP(1) = TOP(I') - 1 \]
\[ I = \text{call } F \Rightarrow \]
\[ \text{if } (F \text{ is an external or indirect}) \]
\[ TOP(I) = \text{zero} \quad \text{(2)} \]
\[ TOP(I) = X_i \quad \text{else} \]
\[ TOP(A) = TOP(I') \text{ where } A \text{ is the first NOP instruction in } F \]
\[ \text{Analyze } F \text{ to get } Y(F) = func(X_1, \ldots, X_n) \]
\[ TOP(I) = TOP(I') + Y(F) \]
\[ I = \text{return from } F \Rightarrow Y(F) = TOP(I') - TOP(A), A \text{ is the first NOP instruction in } F \]
\[ \forall Z = \text{return from } F \Rightarrow TOP(Z) = TOP(I') \]

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

The symbolic equations represented by equations (1) through (3) in figure 2 along with the symbolic unknowns \( X_i \)'s are transformed into a linear system of equations. To solve those equations, we employ our custom linear solver that categorizes 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 \( X_i \)'s 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 \( TOP(I) > 0 \) 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.

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. to identify memory arguments. Surprisingly, very little work is mentioning how to deal efficiently with register arguments and returns as we mention in the introduction section. This 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. 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. 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 and is already done by Second-Write. 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 traverse the executable in reverse call graph order and propagate the set of potential return registers 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 use a simple work list mechanism to handle recursion. 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 are saved into the memory stack, we need a memory analysis technique to track the locations on the memory stack where they are saved. Tracking memory in executables is not a trivial task. Our saved registers detection algorithm 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. \cite{value-set-analysis} 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 efficient in general memory tracking but is sufficient for our application.

As a quick recap 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 bound, an upper bound and a stride.

Before we run the saved registers detection algorithm, we convert the registers inside of each function into the SSA form. Thanks to LLVM, it already has a pass that does that for us. 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 3 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 and quick 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 that address using VSA. At every possible read of that address, we insert a load from that memory location we create. At every possible write to that address, we insert a store to that memory location of the stored value. After that, we run the memory-to-register promotion compiler pass again on those memory locations. Finally,

---

**Algorithm 1:** The callee-saves detection algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Input: A copy of the LLVM IR for a binary</td>
</tr>
<tr>
<td>2</td>
<td>Input: PotArgs : maps functions to their potential register arguments</td>
</tr>
<tr>
<td>3</td>
<td>Input: PotRet : maps functions to their potential return registers</td>
</tr>
<tr>
<td>4</td>
<td>Output: DeadStores : maps functions to the dead register stores</td>
</tr>
<tr>
<td>5</td>
<td>Output: PotRet : The input map after pruning saved registers</td>
</tr>
<tr>
<td>6</td>
<td>foreach reg ∈PotArgs do</td>
</tr>
<tr>
<td>7</td>
<td>Create a dummy register dummy : DummyReg(reg) = dummy</td>
</tr>
<tr>
<td>8</td>
<td>foreach Function F do</td>
</tr>
<tr>
<td>9</td>
<td>foreach Instruction I in F do</td>
</tr>
<tr>
<td>10</td>
<td>if I = store reg, Ptr AND reg ∈ PotArg then</td>
</tr>
<tr>
<td>11</td>
<td>ADDR = ADDR ∪ {(reg,address,X)}</td>
</tr>
<tr>
<td>12</td>
<td>end</td>
</tr>
<tr>
<td>13</td>
<td>if I = store reg, Ptr AND ValueSet(Ptr) ⊇ {address}</td>
</tr>
<tr>
<td>14</td>
<td>ADDR = ADDR ∪ {(reg,address,X)}</td>
</tr>
<tr>
<td>15</td>
<td>end</td>
</tr>
<tr>
<td>16</td>
<td>foreach Instruction I in F do</td>
</tr>
<tr>
<td>17</td>
<td>if I is UnsafeInstruction(address) where (reg,address,X) ∈ ADDR then</td>
</tr>
<tr>
<td>18</td>
<td>insert a volatile load from DummyPtr((reg, address))</td>
</tr>
<tr>
<td>19</td>
<td>end</td>
</tr>
<tr>
<td>20</td>
<td>if I = store reg, Ptr AND ValueSet(Ptr) ⊇ {address}</td>
</tr>
<tr>
<td>21</td>
<td>ADDR = ADDR ∪ {(reg,address,X)}</td>
</tr>
<tr>
<td>22</td>
<td>end</td>
</tr>
<tr>
<td>23</td>
<td>foreach Function F do</td>
</tr>
<tr>
<td>24</td>
<td>if DummyPtr((reg, address)) is deleted AND</td>
</tr>
<tr>
<td>25</td>
<td>DummyReg(reg) has no uses OR only used in return instructions then</td>
</tr>
<tr>
<td>26</td>
<td>DeadStores(F) = DeadStores(F) ∪ {reg}</td>
</tr>
<tr>
<td>27</td>
<td>end</td>
</tr>
<tr>
<td>28</td>
<td>Run LLVM Memory to Register Promotion on All DummyPtr</td>
</tr>
<tr>
<td>29</td>
<td>Run LLVM Dead Code Elimination on F</td>
</tr>
<tr>
<td>30</td>
<td>foreach (reg, address, I) ∈ ADDRS do</td>
</tr>
<tr>
<td>31</td>
<td>if DummyReg(reg) has no uses OR DummyReg(reg) is used in all return instructions of F then</td>
</tr>
<tr>
<td>32</td>
<td>PotRet(F) = PotRet(F) - {reg}</td>
</tr>
<tr>
<td>33</td>
<td>end</td>
</tr>
<tr>
<td>34</td>
<td>end</td>
</tr>
</tbody>
</table>

---
lines [31] through [35] 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 UnsafeUnstruction(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 similar 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 method graph of the executable in reverse order. For each actual 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 [4], the store is not considered a real use. Uses in calls are only considered real uses if the callee takes this 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 argument, there is no need to pass it as an argument at the call site. 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 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. As we explained earlier in section [2] 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 in our algorithm in all compiler generated code where the external function has a standard compiler calling convention; e.g. cdecl, fastcall, thiscall, stdcall and others. Some of the external functions like standard c and c++ libraries are known to SecondWrite, and hence our algorithm will know from the prototypes what registers are needed to be passed. 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 some 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 of our experiments.

Regarding call backs to our rewritten binary, SecondWrite already has a special interrupt handler that redirects control to the corresponding IR function during a callback. We insert assembly code inside the interrupt handler that copies physical registers, as well as stack arguments, at the correct places that are expected for this particular called IR function. This makes things run smoothly if code is recomplied and call backs fire.

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.

The variable and type recovery from executables is a hard problem because unlike source code, symbol tables are absent from executables. 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 already realized this and the best known variable identification technique from executables (DIVINE [7]) is already using an advanced memory analysis technique called value set analysis [6]. DIVINE presents an accurate variable identification technique that is capable of detecting 88% of the memory-allocated variables in executables. The problem with DIVINE is that it is not scalable and requires very long time to complete on small programs. This is our motivation in this work which is to present techniques with the same accuracy and run orders of magnitude faster on larger executables.

One of our contributions in this work is to show that efficient variable detection and type recovery algorithms need not to depend on 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. Variable detection from executables is a best-effort analysis and nobody claims detecting 100% of the variables from executables. 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 rewritten IR while missing some variables due to the unsound pointer analysis comes from the fact that we guarantee that the relative ordering between variables in memory is maintained in the rewritten executable. As an example, if we detect two integer local variables at offsets 0 and 20 on a memory region of size 24 bytes, we will lay out those variables in a structure which has the following three members: 

1. 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]. Laying the variables out in such a structure maintains the correctness of any un-identified memory access to this region. The arrays inserted fill the unknown gaps between variables and maintains 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 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 has the following properties:

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

Having the above relaxations makes our analysis much faster with some sacrifice in precision. The loss of precision is not that much as we will see in our results section. The intuition behind this is as follows: a) Having a flow and context sensitive pointer analysis is not needed for such an application. The reason is that variables usually have the same size and type in all flows and contexts of a program. Some exceptions to this might happen when a variable is a void pointer and is type casted in different flows or contexts to different types. Yet this is not common in programs and happens less often. b) 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. c) Limiting the cardinality of the points-to sets does not affect the precision that much since only few variables will have large points-to sets. 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.

We are able to integrate our variable recovery with a simple yet effective type analysis technique that can type the variables correctly and produces more human-readable and functional IR. Our type recovery technique is based on our best-effort pointer analysis technique. Our type recovery can also detect recursive data types like linked lists and trees. In the following sections, we show the details of our variable and type recovery techniques.

### 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. As we mentioned before, a fast best-effort pointer analysis is needed for our scalable variable recovery. This pointer analysis should be suitable to run on executables where no variables are present. We could have modified current memory analysis schemes on executables like [6] to fit our needs, but we show a simpler analysis that is enough and gives similar variables accuracy to more sophisticated techniques.

Before we begin the analysis, we identify all base memory regions in the executable. Usually, an executable will have three base memory regions. 1) The global memory region where global variables are usually located. Global regions are represented as global sections in the executable. 2) Stack memory region where local variables inside functions are usually 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 as we discussed in section 2. 3) The heap memory region where dynamically allocated variables are usually located. Those are detected 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 [7] in their abstraction. An ALoc contains an offset inside a base memory region and a size representing the variable size. Later we add a type to that representing the variable data type. Variables allocated to registers are represented by IR symbols which represent the SSA form of those registers.

Our best-effort pointer analysis assumes that every detected variable can be a pointer because we do not know data types in the executable and we have to be conservative. We assign point to sets to every IR symbol and detected ALoc. Later 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 subset of the variables on unsized base memory regions instead of totally giving up on them as the case in DIVINE [7]. 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>
<thead>
<tr>
<th>Store $y, x$ (store value $y$ to location $x$)</th>
<th>$\forall z \in \text{ALocs}(\text{PtSet}(x))$: $\text{PtSet}(z) = \text{PtSet}(y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables: UpdateALocs ($\text{PtSet}(x)$, size)</td>
<td>$y = \text{load} x$ $\forall z \in \text{ALocs}(\text{PtSet}(x))$: $\text{PtSet}(y) \cup \text{PtSet}(z)$</td>
</tr>
<tr>
<td>Variables: UpdateScalar ($\text{PtSet}(x)$, size)</td>
<td>$y = x$</td>
</tr>
<tr>
<td>$x = z + z$, PtSet$(z)$ is not empty</td>
<td>$\text{if } z \text{ is a constant then}$ $\text{PtSet}(y) = \text{PtSet}(x) \gg z$ $\text{else}$ $\text{PtSet}(y) = {\top}$</td>
</tr>
<tr>
<td>Variables: UpdateStructure ($\text{PtSet}(x)$, $z$)</td>
<td>$\text{if } z \text{ has SCEV bounds and stride then}$ $\text{UpdateArray ($\text{PtSet}(y)$, stride, bounds)}$</td>
</tr>
</tbody>
</table>

Table 1. Point to sets propagation and variable detection rules
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 newly 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 \[ \text{Table 2} \]. For
a store instruction, the point to sets of the ALocs pointed to by the
pointer operand will be unioned with the point-to set of the value stored. This
is called a weak update in the domain of pointer analysis. A load will set the
load 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 deduce 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. If
the arithmetic is with a non-constant, the resulting points-to set
will be the universal set ‘we reserve bit zero in our bit-vector rep-
resentation for the universal set’. 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 straightforward 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;
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 [6]
can be theoretically used, but will be orders of magnitude slower which makes it less practical in large executables.

Integrating our variable identification system with type recovery
makes the type recovery simpler because it will need only recover
calar 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 
\( \text{setType} \) 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 \( \text{unifyType} \). This function attempts to set the data
type of all the given symbols and ALocs to 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.

The integration between the type analysis and the best-effort
pointer analysis enables us to reveal the shape of some recursive
data structures used and type them correctly. For example; a linked
list allocated on heap consisting of an integer \( \text{key} \) and a \( \text{next} \)
pointer will be typed the following way: variable recovery will reveal that the heap allocation is composed of two ALocs inside
a structure. The \( \text{key} \) variable will be typed according to the rules

\[
\begin{align*}
A &\triangleq \text{call } foo(\text{arg}_1, ..., \text{arg}_n) \\
\text{foo has the known prototype:} & \quad \text{retType}_\text{foo}(\text{type}_1, ..., \text{type}_n) \\
A &\triangleq B \text{ op } C \\
\text{op} &\in \{+, -, \ast, /, \%, >, >=,<, <=\} \\
\text{op has type:} & \quad \text{opType} \\
A, B, C &\text{ has empty points-to sets} \\
A &\triangleq \text{load } B \\
\text{store } A, B &\quad \text{unifyType}(A, \text{ALocs(PSet(B)})) \\
op_1 &\triangleq \text{cast } \text{op}_2, ..., \text{op}_n \\
op_1 &\triangleq \text{cast type } \text{op}_2 \text{ to type} \\
\end{align*}
\]

\[\forall x \in [1, n] \quad \text{setType}(\text{arg}_x, \text{type}_x) \quad \text{setType}(A, \text{retType})\]
in table 5.1 – assuming some instruction exists reveals its type –. The next variable will be declared to be a pointer since its point to set is not empty. We dig into the point to set to discover it is pointing to an address which is declared as the starting point of a structure, so we detect it as a pointer to a structure.

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 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 all SPEC2006 benchmarks which represent 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 some results on those benchmarks without doing comparisons with source code.

The first experiment shows the quality of the recovered variables using the same metrics DIVINE [2] used for comparison purposes. DIVINE [2] 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 are the source code 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 3, an average of 86% of the variables are matched to the debugging information. We run this experiment on programs ranging from 2,149 instructions (me) to 934,292 instructions (gcc). DIVINE [2] reports an average of 88% matched variables on programs ranging between 252 to 5,371 instructions. This shows that our schemes has comparable accuracy to DIVINE [2] 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 8. Our analysis scales linearly with program size for larger binaries. We use all SPEC2006 benchmarks while producing this graph. The analysis takes 14 minutes to analyze games which is a Fortran benchmark whose size is 2.5 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 analyzer that is not guaranteed to have complete points-to sets. We consider that while rewriting the IR to maintain correctness as we discussed earlier in section 5.

The only single SPEC2006 program (out of 28 benchmarks) that did not scale well using our analysis is dealII. The executable for dealII has higher number of indirect memory accesses than other executables. The pointer analysis took longer time to analyze those memory accesses. 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 [17] 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 [17], 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 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 accuracy of multi-level pointers detection. TIE metrics do not show how multi-level pointers are accurately typed since all pointer types have the same height on their lattice [17]. Our accuracy 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 accuracy in this example will be 33%.

Figure 5 shows the distance of our recovered types. Figure 4 shows the conservativeness as well as the accuracy of our detected types. The average distance detected for our type recovery system is 1.7 which is slightly better than the distance of 2 that TIE [17] reports. The conservativeness rate is 96% on average which is slightly higher than 90% that TIE reports. Our accuracy metric shows that we detect 73% of the pointer levels on average. We do not know of any work that reported the accuracy of the recovery of multi-level pointers from executables.

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

The recovered IR after our type analysis is usually of a higher quality than the one before our techniques. To evaluate this, we calculate the percentage of the IR symbols that have a non-generic type after our analysis. Results show that 91% of the IR symbols are typed in Fortran binaries, 88% of them are typed in C and C++ binaries, and 81% of them are typed in the visual studio binaries. Those binaries were compiled using the maximum compiler optimization level while conducting this experiment.

6.2 Decoding the floating point stack

In this section, we show how our techniques are effective in identifying the floating point returns from indirect and external calls. As we mentioned in section, 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 or not. According to our assumptions, whenever we are not sure about an indirect call site or an external call site, we decide conservatively that it is modifying the floating point stack by pushing a single value. We show here how often we took that decision in different binaries. We call that decision the conservative decision.

Figures show the percentage of the unknown calls for which we took the conservative decision in Linux and Windows binaries. 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 float returns for more than 72% of the indirect and external calls on average. We are not aware of any work that identifies such information.

There are three comments one can notice from figures and .

a) The un-optimized binaries tend to have more conservative decisions taken than the optimized ones. This is because non optimized binaries use the floating point stack more heavily than the optimized ones. Usually optimized executables have some of the indirect calls as well as floating point stack variables optimized away which leads to less conservative decisions needed to be taken by us. This is shown more clear in the Fortran executables where they are highly computational – like tonto, leslie, calculix and bwaves –. b) In some exceptional cases, some optimized binaries have more conservative decisions taken – for ex; povray and perl –. Those binaries have many edges in the control flow graph that got removed after doing the compiler optimizations. Those control flow edges were used to derive equations about the stack height, but after optimizations they are not there anymore and hence the equations are removed also. Having less equations leads to more conservative decisions. c) Sometimes the conservative decision rate is very high – as in milc and omnetpp –. Those have higher percentage of indirect calls with very little floating point stack usage and more straight line code than the others. This leads to less number of equations produced and more unknowns that are unresolved.

6.3 Register Arguments and Returns

In this section, we show the accuracy of the detected register arguments and returns. We run our algorithm for all C and C++ benchmarks from SPEC2006 suite 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, the average number of arguments false positives is 0.2 per function. The average number of returns false positives is 0.44 registers per function. These results include the conservative floating point returns we declare from our analysis which translates to why the average number of returns is higher. C++ executables tend to have more indirect calls than C executables which explains why we have more false positives for them.

Comparing this to the work presented in [10], our technique produces more false positives than theirs but on larger set of functions (0.2 false positive arguments in a total of 48,442 functions in our case instead of 0.15 arguments in a total of 13 functions in their work). Their analysis is only capable of identifying a single function at a time. It also requires the program to run multiple times to be able to collect a single function prototype as it is a dynamic analysis. As opposed to our analysis, they cannot guarantee full coverage of arguments and returns because any unused argument or return during an execution trace can be missed. Another main difference is that our analysis takes few seconds to complete on most of our binaries and produces prototypes for all the functions. In their case, they need the same few seconds to only extract a single function (usually less than hundreds of lines of code). This shows that our analysis is more practical than theirs on the expense of some precision loss – by adding more arguments / returns –.

7. Related Work

Throughout the paper, we compared our work with the most recent work done in the areas of variable and type recovery [7] [14] and function prototypes identification [10]. 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 [18], BIRD [20], DynInst [16], DynamoRIO [9]
Many static binary analysis techniques exist that are used to perform alias and dependency analysis [12][15], binary slicing [11], and detect bugs [24]. None of those analysis techniques detect source level variable entries, nor produce a high level IR, C code or an output binary that is functional.

A technique to automatically reconstruct data types from binaries is presented in [13]. It is used in a tool that aims to produce C code from binaries, however no results for actual C code generation are shown. 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. [23] in another attempt to reverse engineer data types in a tool named TyDec for program decompilation. The algorithm expresses the addresses of memory accesses as base + offset; it then reasons about the offsets of equivalent bases.

An early work on type construction from binaries is by Mycroft [19]. It tries to construct C code from binaries with correct type information. However, he does not actually show results on the 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] which produces inline assembly in case it cannot resolve the variables which is not acceptable in our work. 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 as big as two millions instructions.

References