A Stack Memory Abstraction and Symbolic Analysis Framework for Executables

Kapil Anand, University of Maryland, College Park
Khaled Elwazeer, University of Maryland, College Park
Aparna Kotha, University of Maryland, College Park
Matthew Smithson, University of Maryland, College Park
Rajeev Barua, University of Maryland, College Park
Angelos Keromytis, Columbia University

This paper makes three contributions regarding reverse-engineering of executables. First, techniques are presented for recovering a precise and correct stack-memory model in executables while addressing executable-specific challenges such as indirect control transfers. Next, the enhanced memory model is employed to define a novel symbolic analysis framework for executables that can perform the same types of program analyses as source-level tools. Third, a demand-driven framework is presented to enhance the scalability of the symbolic analysis framework. Existing symbolic analysis frameworks for executables fail to simultaneously maintain the properties of correct representation, a precise stack-memory model and scalability. Further, they ignore memory-allocated variables when defining symbolic analysis mechanisms. Our methods do not use symbolic, relocation, or debug information, which are usually absent in deployed binaries. We describe our framework, highlighting the novel intellectual contributions of our approach, and demonstrate its efficacy and robustness. Our techniques improve the precision of existing stack-memory models by 25%, enhance scalability of our basic symbolic analysis mechanism by 10X and successfully uncovers five previously undiscovered information-flow vulnerabilities in several widely-used programs.

Categories and Subject Descriptors: D.3.4 [**Prgramming Languages**]: Code Generation

Additional Key Words and Phrases: Executable code, Program Analysis, Information-flow security

ACM Reference Format:
Kapil Anand, Khaled Elwazeer, Aparna Kotha, Matthew Smithson, Rajeev Barua and Angelos Keromytis, 2016. A Stack Memory Abstraction and Symbolic Analysis Framework for Executables. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2016), 39 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Reverse engineering executable code has received a lot of attention recently in the research community. The demand for advanced executable-level tools is primarily fueled by a rapid rise in zero-day attacks on popular proprietary applications that are available only in the form of executables [FireEye 2015]. Robust reverse-engineering tools are required to completely analyze the impact of the latest cyberattacks on such applications, to define efficient counter strategies and to certify their robustness against such attacks.

Analyzing vulnerabilities directly for binary executables allows rapid response to cyberattacks. Modern threats such as APT3 attacks [FireEye 2015] compromise hosts in a fraction of minutes. Techniques that rely on source code prove to be too slow in responding to such attacks because most users do not have easy access to source code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2016 ACM. 1049-331X/2016/01-ART1 $15.00
DOI: 0000001.0000001

Vulnerability analysis of binary code are likely to enables users to swiftly develop defensive solutions against modern attacks.

Binary executable analysis also enables users to more accurately gauge the criticality of each vulnerability. A binary executable code represents the actual code that runs on a computer. A direct analysis of such a code can enable users to certify whether a vulnerability allows attackers to remotely hijack a system or results in a less severe situation of crashing a program. This aids in prioritizing the defenses against huge number of cyberattacks. A similar analysis of source-code might not enable users to certify the severity of attacks.

Reverse engineering tools are also essential for continuous software maintenance. Various organizations such as US Department of Defense [Darpa 2012] have critical applications that have been developed for older systems and need to be ported to future versions in light of exposed vulnerabilities and in order to effectively exploit features present on modern multicore and parallel systems. In many cases, the source code is no longer accessible thereby engendering a need for advanced tools which enable identification and extraction of procedural components for reuse in new applications.

The applicability of a reverse-engineering framework in such scenarios entails four desired features: 1) The recovered intermediate representation (IR) should be *workable* such that it can be employed to recover an accurate working source code to be analyzed by reverse-engineers and recompiled to obtain a working rewritten binary. A workable representation ensures that the program can be tested and modified using standard debugging techniques. 2) Because executables mainly contain memory locations instead of explicit program variables, the IR should have a *precise memory abstraction* to effectively reason about memory operations by associating each operation with specific memory locations. 3) The framework must support advanced analyses mechanisms on the recovered IR, enabling the *same kind of analysis that can be done on the original source code*. 4) The analysis implemented on recovered IR must be *scalable* to large real-world programs. Unfortunately, presence of executable-specific features such as indirect control transfer instructions (CTI) and lack of any symbolic or debugging information in stripped binary executables make it extremely challenging to conform to such requirements.

This paper makes three main contributions to obtain a reverse-engineering framework that simultaneously meets the above mentioned features. First, we present `MemRecovery`, a hybrid static-dynamic mechanism for recovering a precise stack-memory model and workable IR. Second, we employ this memory model and present a novel symbolic analysis for executables, `ExecSVA`, which enables *analysis similar to source code*. Next, we present `DemandSym`, a demand-driven mechanism to meet the scalability requirements of executable analyses.

`MemRecovery` aims to solve the challenges arising due to executable-specific artifacts such as indirect CTIs in recovering a *precise stack-memory abstraction* while maintaining the *workability* of the IR. A stack-memory abstraction involves associating each stack-memory reference to a set of variables on the memory stack. In order to recover such an abstraction, we need to determine the value of the stack pointer at each program point in a procedure relative to its value at the entry point. This is usually accomplished by analyzing each stack modification instruction in a procedure. Programs typically contain several different calling conventions in which the target procedure may modify the stack to add values because of returns, remove values because of stack cleanup, or keep the stack unmodified. Typically in source code, the programmer must specify the prototype of the target procedure for each indirect call. Hence, a compiler knows the calling convention of the target and its impact on the stack. Unfortunately executable code without any metadata lacks prototypes for procedures. The executable analyzer cannot determine the amount of stack modification due to uncertainty about the calling convention, complicating the task of determining a stack-memory abstraction in the caller procedure after an indirect call.

MemRecovery formulates a set of constraints using control-flow constructs in a procedure to compute the value of stack modification at each call-site inside a procedure. The constraints are solvable in most scenarios. When the constraints cannot be solved, it embeds run-time checks to maintain the workability of IR.

In our second contribution, we employ this memory model and present a novel symbolic analysis for executables, ExecSVA, which enables *analyses similar to source code*. Symbolic analysis [Haghighat and Polychronopoulos 1996; Bodík and Anik 1998] is employed for a variety of applications such as redundancy removal, data-dependence analysis and security analysis. *Symbolic analysis represents the values of program variables as symbolic expressions in terms of previously defined symbols*. The example in Fig 2(a) demonstrates the *symbolic abstraction*[1] resulting from such an analysis.

ExecSVA eliminates the limitations of existing symbolic analysis methods for executables. Existing symbolic frameworks for executables ignore memory models while recovering a symbolic abstraction [Debray et al. 1998; Amme et al. 2000; Cifuentes and Emmerik 2000]. Existing source-level symbolic analysis methods [Haghighat and Polychronopoulos 1996; Bodík and Anik 1998] are inadequate for executables because these methods only focus on instructions involving program variables. ExecSVA computes symbolic expressions for both variables and memory locations, thereby enabling efficient analysis for executables.

Although ExecSVA effectively addresses the concern of memory locations in executables, it faces scalability issues for large programs. It computes a symbolic abstraction over the complete program. As observed in several source-level frameworks, an exhaustive analysis of memory accesses might constrain the *scalability* of the underlying system [Heintze and Tardieu 2001; Tripp et al. 2009; Shankar et al. 2001]. Hence, these systems balance precision and scalability in the presence of pointer operations by employing innovative frameworks such as thin slicing [Sridharan et al. 2007]. Instead of considering all the statements that affect a point of interest, such frameworks only capture the program statements that directly compute the required value.

These existing methods for enhancing scalability rely on several pieces of source-code semantic information and syntactic information that is not available in executables. For example, Sridharan et al. [Sridharan et al. 2007] determine a slice by only focusing on statements updating the corresponding fields of the same data structure. The lack of such structural information in executables limits the application of this method to executables.

In our third contribution, we present DemandSym, a demand-driven mechanism, to further enhance the scalability of ExecSVA for executables. Instead of doing an exhaustive analysis over the complete program, DemandSym first computes the set of program objects critical for enforcing a particular client analysis (for example, information-flow violations) and henceforth computes symbolic abstraction for only these objects.

Our techniques are highly effective in meeting the above mentioned goals of reverse-engineering frameworks. MemRecovery improves the precision of memory models by 25% in programs containing a significant number of indirect CTIs. The criterion for significant number of indirect CTIs is presented in Section 8. ExecSVA without the demand-driven enhancement is somewhat scalable and analyzes large programs such as *gcc* (250,000 lines of code) in around 11 minutes. DemandSym achieves a 10x improve-

[1]In literature, symbolic abstraction has another usage. As presented by Reps et al. [Reps et al. 2004], symbolic abstraction is described as the best value of a particular formula in a given abstract domain that over-approximates its meaning. In current paper, symbolic abstraction refers to the abstraction obtained by ExecSVA.

ment in the scalability of `ExecSVA` and analyzes large programs such as *gcc* within a minute. We extend our analysis for several applications such as security analysis and program parallelization. We demonstrate that client applications become less effective when memory tracking is not enabled. Our extensions for information-flow analysis uncover five previously undiscovered vulnerabilities in popular file transfer and internet relay chat programs.

2. MOTIVATION

In this section, we demonstrate limitations of existing binary executable frameworks in obtaining a workable IR with a precise stack-memory model, the relative importance of considering a memory model for symbolic abstraction, and the efficacy of our demand-driven framework in enhancing the scalability of symbolic abstraction computation. We present several examples to demonstrate the limitations of existing binary analysis frameworks and establish a requirement of novel techniques in order to achieve the goals mentioned in Section 1. In several scenarios, we also present how existing source-level techniques would perform on executables if they are applied to executables without any modification. These source-code techniques will work perfectly when they are applied to source-code. This discussion is presented only to motivate the fact that source-level techniques need to be adapted properly before applying them to binary executables. The following sections will describe in detail our method of adapting such techniques to binary executable code.

Precise and correct stack-memory abstraction: A source program has an abstract stack representation where the local variables are assumed to be present on the stack but their precise layout is not specified. In contrast, an executable has a fixed physical stack layout.

To recreate an intermediate representation (IR), the physical stack must be deconstructed to individual abstract frames per procedure. Since each such frame contains variables from the source code, a memory model is defined as precise if each frame can be divided into abstract locations analogous to the original variables.

Previous methods [Balakrishnan and Reps 2004] have approached this problem in two steps. First, all the instructions in a procedure that can modify the stack pointer are analyzed to compute the maximum size to which the stack can grow in a single invocation of the procedure. Next, each such frame is further abstracted through a set of `a-locs`. An `a-loc` is characterized by two attributes: its relative offset in the region with respect to other `a-locs`, and its size. The `a-loc` representation requires the value of the stack pointer to be determined at each program point in a procedure relative to its value at the entry point.

As highlighted in Section 1, this is usually accomplished by tracking each update to the stack pointer. However, several artifacts might result in a non-deterministic stack modification, invalidating the inherent assumption in previous frameworks [Balakrishnan and Reps 2004]. We characterize the impact of a control transfer instruction (CTI) `I` on the value of the stack pointer using the following definition:

`StackDiff(I)` = Stack Pointer after `I` − Stack Pointer before `I`.

The term `StackDiff` can be applied to either a CTI or the procedure called by the CTI, and represents the stack modification amount in either case. The `StackDiff` of a CTI is positive if the called procedure cleans up its arguments, or zero if it does not. In theory, it can be negative if the procedure leaves some local allocations on the stack, although we have not observed this in compiled code. Previous approaches calculate the value of `StackDiff` by symbolically evaluating all the stack modification instructions in a procedure [Balakrishnan and Reps 2004]. As per these methods, `StackDiff` at an

```
main:
1   sub 24, $esp          //Local Allocation
2   mov $10, 8(%esp)      //Access (%esp+8)
3   call *%eax            // An Indirect call
4   mov $20, 12(%esp)    //Access
                         //(%esp+12+UNKNOWN)
      ......
```

Fig. 1: *An example demonstrating the imprecision in the presence of indirect calls; second operand in an instruction is the destination.*

indirect CTI is deterministic if all possible targets have the same value of `StackDiff`. Thereafter, the stack pointer in the caller procedure is adjusted by `StackDiff` amount. This adjustment is imperative for maintaining the correctness of data-flow.

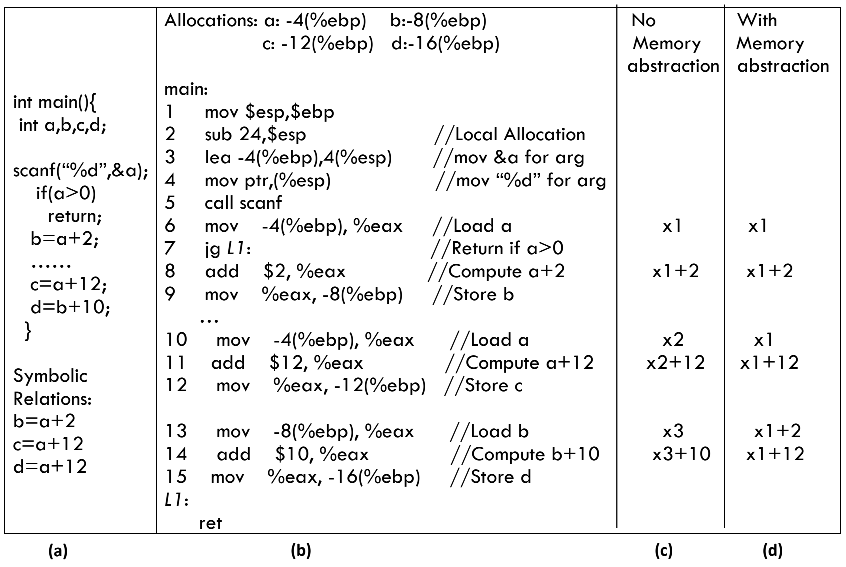
However, `StackDiff` cannot be determined statically in all scenarios. For example, the possible targets of an indirect CTI might have different `StackDiff`, or an external function with an unknown prototype might have a statically unknown `StackDiff`. In such cases, existing frameworks either result in an imprecise memory abstraction or fail to maintain the correctness. CodeSurfer/X86 just issues an error report if it cannot determine that the change is a constant [Balakrishnan and Reps 2004]. IDAPro, a widely-used binary analysis tool, applies a constraint-based mechanism to compute the values of `StackDiff`. However, when the underlying method fails to determine a unique solution, it compromises the correctness of the output IR by accepting one feasible solution (which could be wrong) out of a potentially very large number of possible outcomes [HexBlog 2006].

Figure 1 illustrates an example of a scenario where `StackDiff` cannot be determined statically. In Fig. 1, a local region of size 24 is allocated in a procedure at Line 1. Consequently, the memory access at Line 2 results in the discovery of an `a-loc` at offset `16`. Suppose the possible targets of the indirect CTI at line 3 have different `StackDiff` values. Consequently, `esp` after Line 3 has an unknown offset relative to its value at the entry point of the procedure. Hence, no `a-loc` can be identified at Line 4. On the other hand, if existing tools such as IDAPro choose a wrong value of `StackDiff` from the set of possible solutions, it results in an incorrect data-flow at Line 4. For example, if the actual value is 0 while existing tools assign a value of -4, then the output representation will include a wrong data flow edge from Line 2 to Line 4.

Our hybrid mechanism, `MemRecovery`, achieves precision in the identification of stack `a-locs` as well as the workability of the recovered IR. Our static mechanism enables memory abstraction through a set of `a-locs` while our dynamic mechanism guarantees correctness of the IR when `StackDiff` cannot be computed.

Symbolic abstraction: Executables extensively employ memory locations, hence, not analyzing them for symbolic analysis results in imprecise symbolic relations. Figure 2(a) shows a source code example and the relations between various computations determined through symbolic analysis. Figure 2(b) shows a sample code that might arise when the example in Fig. 2(a) is converted to an executable. Here, variables `a`, `b`, `c` and `d` are allocated to memory locations. Fig. 2(c) shows symbolic relations obtained by applying previous symbolic analysis techniques for executables [Debray et al. 1998] on the code in Figure 2(b). *Since these techniques do not propagate symbolic expressions across memory locations, a new symbol is defined at every memory reference instruction*. As evident, the resulting symbolic relations are conservative and yield imprecise program information.

Similar imprecise results are obtained if existing symbolic analyses for source code [Haghighat and Polychronopoulos 1996] are applied directly to the above code without any modification. Symbolic analysis techniques for source-code also propagate



	Allocations: a: -4(%ebp) b:-8(%ebp)	No	With
	c: -12(%ebp) d:-16(%ebp)	Memory	Memory
		abstraction	abstraction
int main(){	main:		
int a,b,c,d;	1 mov $esp,$ebp		
	2 sub 24,$esp //Local Allocation		
	3 lea -4(%ebp),4(%esp) //mov &a for arg		
scanf("%d",&a);	4 mov ptr,(%esp) //mov "%d" for arg		
if(a>0)	5 call scanf		
return;	6 mov -4(%ebp), %eax //Load a	x1	x1
b=a+2;	7 jg *L1*: //Return if a>0		
......	8 add $2, %eax //Compute a+2	x1+2	x1+2
c=a+12;	9 mov %eax, -8(%ebp) //Store b		
d=b+10;	...		
}	10 mov -4(%ebp), %eax //Load a	x2	x1
Symbolic	11 add $12, %eax //Compute a+12	x2+12	x1+12
Relations:	12 mov %eax, -12(%ebp) //Store c		
b=a+2	13 mov -8(%ebp), %eax //Load b	x3	x1+2
c=a+12	14 add $10, %eax //Compute b+10	x3+10	x1+12
d=a+12	15 mov %eax, -16(%ebp) //Store d		
	L1:		
	ret		
(a)	(b)	(c)	(d)

Fig. 2: *(a) A sample C code (b) Corresponding assembly code; the second operand in an instruction is the destination (c) Symbolic relations on the assembly code with no memory abstraction (d) Symbolic relations on the assembly code with memory abstraction.*

symbolic relations only across variables because such techniques are typically applied before register allocation. Hence, these techniques need to be adapted before applying them to executables.

We observe that representing a symbolic abstraction for memory locations can eliminate this limitation. Figure 2(d) shows the symbolic relations when the abstraction is maintained for memory locations as well. Suppose the variable `a` (`-4(%ebp)`) has value `x1` in the environment of symbolic abstraction. Hence, the representation of symbolic abstraction for memory locations implies that the variable `%eax` at Line 6 and Line 10 is assigned value `x1`. Similarly, the memory location `-8(%ebp)` at Line 9 and the variable `%eax` at Line 13 are assigned value `x1+2`. Propagation of these values results in symbolic relations that are similar to those obtained for the original source code.

Demand-driven computation: `ExecSVA` framework computes symbolic abstraction for all the program objects irrespective of the target client analysis for which it is employed. In several scenarios, such computation can be made more scalable by adapting it to a client analysis. We demonstrate this potential for scalability using information-flow violations as the target analysis.

Information flow violations collectively comprise one of the most critical security vulnerabilities [Dalton et al. 2007]. Such violations subject the programs to severe security attacks, such as format-string attacks [Shankar et al. 2001], directory-traversal attacks [Dalton et al. 2007] and SQL-injection [Tripp et al. 2009].

Various techniques, under the broad umbrella of taint propagation, have been proposed to detect such information-flow violations. The basic idea is to *taint* the inputs coming from an untrusted source, propagate the *taint* through the data and control flow of the program and check if the *taint* ever reaches a sensitive sink without flowing through a sanitization routine.

The symbolic abstraction computed through `ExecSVA` effectively captures the flow of information between two locations in an executable while countering executable-

```
    typedef struct temp{
    int x;
    int y;
    }temp;

1.  int main(){
2.  temp* z1, *z2;
3.  temp* x1 =
    malloc(sizeof(temp));
4.  z1 = x1;
5.  z2= x1;
6.
7.  (*z1). x = SOURCE();
8.
9.  if(z1==z2)
10. {
11.
12.       SINK((*z2).x);
13. }
14. }
```

(a) Pseudo source code snippet

```
1.  main:
2.  subl $32, %esp
3.  %eax = call malloc
4.  store  %eax, 28(%esp)
5.  store  %eax, 24(%esp)
6.
7.  %ebx = load 28(%esp)
8.  %ecx = call SOURCE()
9.  store %ecx, (%ebx)
10.
11. cmpl  24(%esp), %ebx
12. jne  L2
13.
14. %ecx = load 24(%esp)
15. %edx = load (%ecx)
16. call SINK(%edx)
17. L2:
```

(b) Pseudo assembly code snippet

Fig. 3: *A small program to illustrate direct data dependent statements, shown as underlined.*

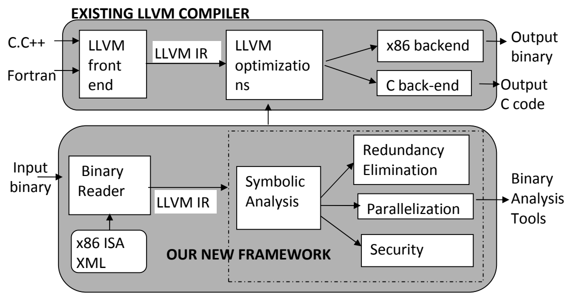


Fig. 4: *Organization of the system.*

specific challenges such as lack of variables and procedure prototypes. `ExecSVA`, therefore, provides an effective platform for implementing taint propagation in executables.

As mentioned in Section 1, several source-code frameworks employ mechanisms such as thin slicing to enhance the scalability of taint propagation. As we described above, such source-code techniques cannot be directly applied to executables. Figure 3(a) shows a simple source-code example containing a potentially unsafe flow from an untrusted source to a sensitive sink. Figure 3(b) shows an assembly code corresponding to the source-code in Fig. 3(a). The program statements which are most relevant for determining the unsafe flow of information to the sensitive sink are underlined in Fig. 3(a), which constitute the thin slice for this sink. The thin slice is computed by detecting that the same field of a data structure is accessed at Line 7 and Line 12 [Sridharan et al. 2007]. As evident, the field information is not present in the assembly code due to loss of syntactic information during the compilation process. Consequently, a direct application of above thin slicing method without any modification will not capture the statement at Line 9 in Fig. 3(b).

We formulate a set of rules which enable us to compute the set of direct data dependent statements in executables, similar to existing thin slicing mechanisms in source-code. In Fig. 3(b), the underlined program statements are the set of statements which are considered for computing the symbolic abstraction at Line 16. This obviates the need of computing symbolic abstraction for remaining data-objects in this program.

3. OVERVIEW OF OUR FRAMEWORK

Figure 4 presents an overview of our binary analysis framework. Our framework is built over the existing SecondWrite framework [Anand et al. 2013a]. SecondWrite translates the input x86 binary code to a workable program represented in the intermediate representation (IR) of the LLVM Compiler [Lattner and Adve 2004]. SecondWrite obtains an IR that contains features such as procedures, procedure arguments and return values. *This conversion back to a compiler IR is not a necessity for the work we present; any binary system including IDAPro [IDAPro disassembler] and Code-surfer [Balakrishnan and Reps 2004] can use our analysis*. LLVM IR recovered by SecondWrite is passed through our analysis system.

The binary reader module in SecondWrite implements several mechanisms [Smithson et al. 2013] to address code discovery problems in executables and to handle indirect control transfers. Here, we briefly summarize these mechanisms.

A key challenge in executable frameworks is discovering which portions of the code section are definitely code. SecondWrite [Smithson et al. 2013] implements *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 is based on an assumption that an indirect control transfer instruction (CTI) requires an absolute address operand, and that these address operands should appear within the code and/or data segments. This assumption holds true in compiled code unless it is generated to be position-independent. The code and data segments are scanned for values that lie within the range of the code segment. Under the assumptions of compiled code without any hand-coded assembly, the resulting values contain all of the indirect CTI targets. The detailed set of assumptions behind this model are discussed in Section 7.

Indirect CTIs arising due to switch statements in source-code also adhere to the above restriction. Compilers typically use jump tables to implement dense switch statements. The entry point address of each case is stored in a table and the input value is used as an index into the table of absolute addresses. Although the address used to access the table is calculated at runtime, it does not present a problem, as this is a data reference. Importantly, the indirect control-flow target addresses are all statically calculated. Smithson et al [Smithson et al. 2013] provide detailed description of handling other kinds of control transfers as well.

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

4. RECOVERING PRECISE STACK-MEMORY MODEL

In this section, we discuss `MemRecovery`, our hybrid static-dynamic solution for obtaining a workable representation with a precise stack-memory model. We first present a symbolic constraint mechanism to determine the value of `StackDiff` for each CTI where it is unknown. Next, we discuss our solution for maintaining the workability of recovered IR even when `StackDiff` at some CTIs cannot be solved.

In case of a direct CTI, `StackDiff` of the CTI is equivalent to `StackDiff` of the called procedure because each direct CTI is associated with a unique determinable procedure. There is no such direct association in case of indirect CTI because an indirect CTI might call several different procedures at runtime.

Our analysis employs `StackDiff` values of procedures for computing `StackDiff` for indirect CTIs. The prototypes of well-known library functions, similar to the IDAPro's FLIRT database [IDAPro disassembler], are employed for determining their `StackDiff` value. Library functions whose prototypes cannot be found are handled correctly using our method of determining unknown `StackDiff` values. We assume that existing methods [Balakrishnan and Reps 2004] are able to determine the value of `StackDiff` for each procedure, which holds true under the assumptions of *standard compilation model* [Balakrishnan and Reps 2004]. A standard compilation model assumes that each procedure may allocate an optional stack frame in only one direction and each variable resides at a fixed offset in its corresponding region. This assumption is explained in more detail in Section 7.

4.1. Static Computation

A CTI `I` can result in an unknown `StackDiff` in three cases, which we collectively refer to as *Unknown CTIs*.
Case 1: `I` is a direct CTI to an external procedure with unknown prototype.
Case 2: `I` is an indirect CTI with unresolved targets.
Case 3: `I` is an indirect CTI and its targets have different `StackDiff` values.

In such scenarios, our mechanism employs several boundary conditions imposed by the control-flow inside the corresponding caller procedure to determine `StackDiff`. The proposed constraint formulation does not require us to determine the precise set of targets of an indirect CTI, which itself is an undecidable problem.

We define symbolic values X_I for representing `StackDiff` of CTI `I` and S_I for representing the local stack height at program point before `I`. Every stack modification instruction in a procedure is analyzed to derive an expression of S_I in terms of the X_Is. The resulting expressions are transformed into a linear system of equations that can be solved to calculate X_Is.

Figure 5 presents the rules for generating symbolic constraints and equations in a particular procedure `P`. It presents rules for analyzing each stack modification instruction, a set of initialization and boundary conditions for solving the symbolic equations and a set of conditions which invalidate our symbolic constraints for the current procedure.

In an x86 program, several instructions can modify the value of stack pointer. The local frame in a procedure is usually allocated by subtracting a constant value from `esp`. Similarly, the local frame is deallocated by adding a constant amount to `esp`. `Push` and `pop` instructions implicitly modify the stack pointer by the size of amount pushed onto the stack. The rules in Fig. 5 incorporate the deterministic modification at each CTI. An indeterminate modification is modeled symbolically as X_I. The dataflow rules in Fig. 5 obtain an expression for S_I considering each such stack modification instruction.

In order to solve the above symbolic equations, Fig. 5 generates two constraints based on the control-flow in procedure `P`. These conditions hold true for every executable following the standard compilation model [Balakrishnan and Reps 2004]:

→ $\forall \texttt{Pred} \in \texttt{Pred}_{\texttt{BB}}, S_{\texttt{BeginBB}} = S_{\texttt{EndPred}}$: This condition implies that at a merge point in the control-flow of a procedure, the stack height at the end of every predecessor basic block must be equal. Otherwise, any subsequent stack access might access different stack locations depending on the path taken at run time, resulting in an indeterminate behavior.

→ $S_I = 0 \; \forall \; \texttt{ret} \in \texttt{P}$: In an x86 program, a return instruction loads an address from the location pointed by `esp` and sets the program counter to the loaded value. The return address is pushed to the stack by the caller procedure and a compiled program (without any hand-coded assembly) usually accesses the return address location only through a return instruction. It does not access or modify this location directly through any other instruction. Hence, `esp` can refer to the return address only if stack height S_I is zero. Thereafter the return instruction may optionally specify an operand to clean up incoming arguments, so `StackDiff` could be positive or zero.

Figure 5 also formulates the following conditions which invalidate the assumptions behind our boundary conditions. In such situations, we discontinue our static mechanism and rely on our dynamic mechanism to maintain correctness.

Unknown Symbolic Values : X_I, where X_I = `StackDiff` of procedure call `I`

Initial/Helper Variables :

`Targ(T)`: Set of procedures targeted by call target operand T

`StackDiff(f)`: StackDiff of procedure `f`; `Y_SET(F)` = $\{$`StackDiff(f) : f` $\in$ `F`$\}$

`Begin(P)` = Entry point of procedure `P`; `Pred(BB)` = Predecessors of basic block `BB`;

`Begin(BB)`,`End(BB)` = Entry point,terminator of basic block `BB`

S'_I = Stack height after instruction `I`; S_{BB} = Stack height at beginning of basic block `BB`;

`Prev(I)` = the previous instruction to `I` (`I` $\neq$ `Begin(BB)`)

S_I = if (I $\neq$ `Begin(BB)`) then $S_{Prev(I)}$ else S_{BB}

`R` : A register, `Size(R)`: Size of register `R`, `N`: A constant

Initial Conditions : $S_{Begin(P)} = 0$

Data flow rules :

```
For every instruction I:
  I = push R ⇒ $S'_I = S_I + size(R)$
  I = pop R ⇒ $S'_I = S_I - size(R)$
  I = add esp, N ⇒ $S'_I = S_I - N$
  I = sub esp, N ⇒ $S'_I = S_I + N$
  I = jmp L ⇒ $S'_{Begin(L)} = S_I$
  I = call Y ⇒
    if (Y_SET(Targ(Y)) contains a single constant C)
      $S'_I = S_I + C$
    else
      $S'_I = S_I + X_I$
  default (if not an invalidation condition) ⇒ $S'_I = S_I$

  Special Rules:
  I = mov esp, ebp ⇒ $S'_I = 0$
  I = leave ⇒ $S'_I = 0$
```

Boundary Conditions :

1. $\forall$ BB: $\forall$ P $\in$ `Pred(BB)`, $S_{Begin(BB)} = S_{End(P)}$
2. `I` = ret : Constraint $S_I = 0$

Invalidation Conditions :

1. `I` = `esp` $\leftarrow$ ... /* Any assignment except in data-flow rules */
2. `I` accesses return address

Fig. 5: *Data flow rules used to determine stack modifications in a procedure P.*

$\rightarrow$ `I` = `esp` $\leftarrow$ ... : Any assignment to `esp` other than those in data-flow rules implies a local frame allocation of variable size. In such a scenario, the boundary conditions fail to obtain a solution for X_I. However, this condition arises in extremely rare circumstances of variable size arrays on stack frame. We make two explicit exceptions to the above constraint to handle common compilation procedure. Several compilers such as `gcc` employ a frame pointer to save off the stack pointer at the beginning of a procedure and restore the stack by assigning the saved value back to `esp`. Similarly, several code segments contain x86 idioms such as `leave` instruction which implicitly assign a previously stored value to `esp`. This idiom is generally used in restoring the value of the stack pointer that was saved on the stack at the entry on the program. Even though these two scenarios violate this constraint, we model these two instructions as setting S_I to zero.

$\rightarrow$ `I` accesses return address: In a usual compiled code, `StackDiff` is either zero or positive. In theory, procedures could have a negative `StackDiff`, implying that the procedure leaves some local allocations on the stack. In such scenarios, `esp` would not point to the return address at the point of return. Hence, a return must be implemented by

```
main:
ESP = ALLOCA 24                    //Local Allocation
8(ESP) = 10                        //Access (%esp+8)
stackDiff = call CallTrans(EAX)    // An Indirect call
ESP = ESP + stackDiff              //StackDiff Adjustment
12(ESP) = 20                       //Access
                                   //(ESP+12+UNKNOWN)
```

Fig. 6: *An example demonstrating the insertion of runtime check for computing stackDiff in presence of an indirect control transfer instruction, second operand in the instruction is the destination.*

explicitly accessing the return address from the middle of the stack. This invalidates the assumption behind our boundary condition 2 and we resort to run-time checks.

The resulting symbolic equations are solved by employing a custom linear solver that categorizes the equations into disjoint groups based on the variables used in every equation. A group is solved 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. Once we obtain a solution of X_I for each I in a procedure, we can obtain a safe abstraction of memory regions into a set of *a-locs* using the methods in [Balakrishnan and Reps 2004].

Wazeer et al [ElWazeer et al. 2013b] present symbolic equations based framework to obtain a solution for an orthogonal problem of recovering floating point variables from floating point stack in x86 architecture. x86 machine code maintains floating point variables in the form of a stack. These two methods might appear similar since they employ the same underlying tool of symbolic equations but they have a number of fundamental differences. For instance, these two methods make inferences about usage of stack, but the above method determines the value of stack pointer at each program point relative to its value at entry point of procedure while Wazeer et al. present a method to determinate the top of floating point stack at every program point. These two methods differ significantly in the kind of invalidation conditions that can be applied. Wazeer et al. [ElWazeer et al. 2013b] argue that assuming a fixed solution is sufficient to guarantee the accuracy in floating point stack. Hence, if a solution cannot be determined through linear equations, Wazeer et al. assume one fixed solution out of many possible solutions. On the other hand, we provide a runtime solution to handle memory stack, as discussed next, to represent the scenarios accurately when solution cannot be determined. In addition, Wazeer et al. only analyze instructions that modify floating point stack (such as floating point push and pop) while the above method present analysis for all the instructions that modify memory stack (such as push, pop, sub, add etc).

4.2. Dynamic Mechanism

As mentioned above, the above method does not guarantee a solution for all the scenarios. For example, it fails to determine the value of `StackDiff` in basic blocks containing multiple CTIs each with an unknown X_I value. Below, we discuss our dynamic mechanism to correctly represent all the three cases of *Unknown CTIs* as part of a workable IR.

An explicit representation of `StackDiff` is imperative to maintain the correctness of the IR. Recall from Section 2, the physical stack is deconstructed into individual abstract frames per procedure in our IR. Hence, all the modifications to stack must be explicitly represented in the procedure body, otherwise it will lead to incorrect dataflow representation in IR. This includes the modification to stack as a result of CTIs.

```
Sym := Sym+T|T
T :=T*F|F
F := I | n
I := [IR Variables]
n := [Int]
```

Fig. 7: *Grammar for symbolic expressions. + and * are standard arithmetic operators, Int is the set of all integers, IR Variables are symbols in the obtained intermediate representation. IR Variables and integers constitute symbolic alphabets in this grammar.*

Our dynamic mechanism statically inserts checks in the IR that determine `StackDiff` value at runtime when IR will be employed for rewriting and debugging. The correct representation of such scenarios does not directly aid static analysis. An unknown modification to stack results in conservative treatment of subsequent stack access instructions in the procedure. However, such a representation fulfills our goal of obtaining a workable IR.
Case 1: Since this case represents control transfer to an external procedure, the body of the called procedure cannot be modified. Such scenarios are represented in IR through a trampoline for calling the external procedure. The trampoline computes the shift in stack pointer value before and after the call using inline assembly instructions.
Case 2 and **Case 3**: Recall from Section 3, an indirect CTI is translated to the corresponding location in IR using a switch statement inside a *call translator* procedure. In such scenarios, `StackDiff` is declared as an explicit return variable in the call translator procedure. The definition of the call translator is modified to return the value of `StackDiff` for the called procedure in each switch statement. The corresponding callsite is updated by adjusting the stack pointer value with returned value of `StackDiff`. Figure 6 shows an example where callsite of an indirect control transfer in Fig. 1 is updated with `StackDiff` value. Line 4 in Fig. 6 shows the modification of the local stack pointer with `StackDiff` value returned by the call translator procedure. As evident, such a modification does not enhance the static analysis of subsequent stack accesses (Line 5) but ensures the correctness of resulting IR.

5. SYMBOLIC VALUE ANALYSIS

Our technique for symbolic analysis for executables, `ExecSVA`, is a flow-sensitive, context insensitive analysis which computes an approximation of a set of symbolic values that each data object (variables and `a-locs`) can hold at each program point. `ExecSVA` defines three kind of memory regions, associated with procedures (`Stack`), global data (`Global`) and heaps (`HeapRgn`). Each memory region is further abstracted through a set of `a-locs` (defined in Section 2) using methods proposed by Balakrishnan et al. [Balakrishnan and Reps 2004]. `MemRecovery` mechanism enhances the precision of such `a-locs` associated with `Stack` memory region. VSA is also employed as an underlying pointer analysis mechanism for our analysis.

5.1. Symbolic Abstraction

Figure 7 presents the grammar for representing the symbolic expressions in our abstraction. As evident from Fig. 7, symbols in the obtained IR and integers constitute *symbolic alphabets* in this grammar. Symbolic expressions are numeric algebraic polynomials containing sums of product terms of variables. The grammar is designed to capture most common computational instruction patterns present in an x86 binary executable such as *add, mov, sub and mul* [Kankowski 2006]. The presence of multiplication inside addition captures common pointer arithmetic expressions such as *base + scale*index* present in a binary executable. Grammar based on most common instructions ensures a balance between precision and efficiency of resulting symbolic representation.

`Symbolic Value Set`: A symbolic value set is a finite set of symbolic expressions defined by the Grammar in Fig. 7. It constitutes an approximation of the set of symbolic values that each data object holds.

The abstraction supports standard arithmetic set operators such as Addition ($\oplus$) and Multiplication ($\otimes$) as well as a widen (∇) operators. These operators are defined below:

1. Add Operator ($\oplus$): This operator computes a new symbolic value set by adding each symbolic expression present in $SymValSet_2$ to each symbolic expression present in $SymValSet_1$.

$$SymValSet_1 \oplus SymValSet_2 = \{sym1 + sym2 \mid sym1 \in SymValSet_1, sym2 \in SymValSet_2\} \tag{1}$$

2. Multiply Operator ($\otimes$): This operator computes a new symbolic value set by applying the multiplication operator between each symbolic expression present in $SymValSet_1$ and $SymValSet_2$:

$$SymValSet_1 \otimes SymValSet_2 = \{sym1 * sym2 \mid sym1 \in SymValSet_1, sym2 \in SymValSet_2\} \tag{2}$$

3. Widen operator (∇) : This operator is defined to ensure fast convergence of our analysis. If the required cardinality of a symbolic value set increases beyond a limit, we invalidate the symbolic value set. This operation ensures that the analysis converges in a limited interval of time and is scalable to large applications.

$$\nabla \mathtt{SymValSet}_1 = \{\mathtt{if}\ |\mathtt{SymValSet}_1| > \mathtt{LIMIT}, \mathtt{then}\ \top \quad \mathtt{else}\ \mathtt{SymValSet}_1\} \tag{3}$$

5.2. Intraprocedural Analysis

Our method assumes that the symbols corresponding to the binary code's registers have been converted to single-static assignment (SSA) form before running our analysis. Since in SSA form each variable is assigned exactly once, a single symbolic map is sufficient to maintain flow-sensitive symbolic value sets for variables. However, memory locations are usually not implemented in SSA format in IR. Consequently, a symbolic map is maintained at each program point to represent flow-sensitive symbolic value sets for memory locations. Hence, symbolic value analysis effectively computes the following maps:

`SR`: Map between `Vars` and their corresponding symbolic value sets.
$\mathtt{SM}_{\mathtt{I}}$: Map between `a-locs` and their corresponding symbolic value sets at a program point before an instruction `I`

Executables regularly employ the *indirect-addressing* mode for accessing memory locations. VSA [Balakrishnan and Reps 2004] is employed to determine the set of memory addresses which each direct or indirect memory access instruction can access. Given a set of `a-locs`, VSA can compute an over-approximation of the set of `a-locs` that each register and each `a-loc` holds at a particular program point.

The algorithm is implemented on the IR, but is presented on C-like pseudo instructions for ease of understanding. Each instruction in the IR implements a transfer function which translates the symbolic maps defined at its input to the symbolic maps at its output. The following definitions are introduced to ease the presentation.

$\mathtt{R_i}$: IR (SSA) variables
r: Data object (Var or a-loc)
$\mathtt{SM'_I}$: Map between a-locs and their symbolic value sets after Instruction I
SR(r): Mapping of Var r in map SR
$\mathtt{SM_I(r)}$: Mapping of a-loc r in map $\mathtt{SM_I}$
$\mathtt{Mem_I(r)}$:Set of memory addresses that r can hold at program point before Instruction I (obtained by VSA)
(r,SV): Pairing between a data object r and a symbolic value set SV

The memory abstraction includes a concept of *fully accessed* and *partially accessed* a-locs. In order to understand partial a-locs, consider that $\mathrm{Mem}_I(r)$ contains a list of memory addresses that the data object r can hold at program point before Instruction I. If this object is dereferenced in a memory access instruction of size s, the a-locs, that are of size s and whose starting addresses are in set $\mathtt{Mem_I(r)}$, represents the fully accessed a-locs. The partially accessed a-locs consists (i) a-locs whose starting addresses are in $\mathtt{Mem_I(r)}$ but are not of size s and (ii) a-locs whose addresses are in $\mathtt{Mem_I(r)}$ but whose starting addresses and size do not meet the condition to be fully accessed a-locs. Using the notation from [Balakrishnan and Reps 2004], this operation is mathematically represented as:

$\{\mathtt{F,P}\} = *(\mathtt{Mem_I(r)},s)$

Here, F represents the fully accessed and P represent the partially accessed a-locs. As the name suggests, only some portion of a partial a-loc is updated or referenced in a memory access instruction. Hence, they are treated conservatively in our analysis, as will be explained below.

Table I shows the mathematical forms of transfer functions for each instruction. Below, each of these transfer functions is discussed in detail.

1. Assignment: `e: R1 := R2`
This is the basic operation where symbolic analysis behaves similarly to the concrete evaluation. Any existing entry in the symbolic map SR corresponding to the variable R1 (computed in an earlier iteration) is removed from the map and the symbolic value set of variable R2 is assigned to variable R1.

2. Arithmetic Operation: `e: R3 := R2 OP R1`
In such scenarios, the analysis evaluates the symbolic values according to the underlying mathematical operator. The evaluation is defined for addition, subtraction and multiplication operators. Addition and multiplication are handled by employing the underlying ($\oplus$) and ($\otimes$) operators respectively. Subtraction operation is handled analogous to the addition by reversing the sign of each coefficient in the symbolic expressions of second operand, R1. Since the remaining operations are not represented, a new symbolic expression is introduced to represent the result of the computation.

3. Memory Load `e: R1 := *(R2)`
The analysis relies on obtaining the *a-locs* accessed by this instruction. If the current instruction does not access any partial a-loc, the symbolic value of variable R1 is computed by unioning the symbolic values corresponding to each of the possible a-loc. Otherwise, it is assigned $\top$.

4. Memory store `e: *(R2) := R1`
The propagation of symbolic values is governed by current memory store accessing a single a-loc or multiple a-locs. If the current memory store only updates a single

Name	Operation	Transfer Function
1. Assignment	$R1 := R2$	$SR = \{SR - SR(R1)\} \cup \{(R1, SR(R2))\}$
2. Arithmetic	$R3 := R2\, OP\, R1$	$\begin{array}{l} if\; OP = + \\ \quad tmp = \nabla(SR(R2) \oplus SR(R1)) \\ if\; OP = - \\ \quad tmp = \nabla(SR(R2) \oplus ((-1) * SR(R1))) \\ if\; OP = * \\ \quad tmp = \nabla(SR(R2) \otimes SR(R1)) \\ else \quad //Create\; a\; new\; symbolic\; expression \\ \quad tmp = R3 \\ SR = \{SR - SR(R3)\} \cup \{(R3, tmp)\} \end{array}$
3. Load	$R1 := *(R2)$	$\begin{array}{l} \{F, P\} = *(\mathrm{Mem}_e(R2), s) \\ if\; \|P\| = 0 \\ \quad tmp = \nabla(\bigcup_{v \in F} SM_e(v)) \\ else \\ \quad tmp = \top \\ SR = \{SR - SR(R1)\} \cup \{(R1, tmp)\} \end{array}$
4. Store	$*(R2) := R1$	$\begin{array}{l} \{F, P\} = *(\mathrm{Mem}_e(R2), s) \\ if\; \|F\| = 1\; \&\; \|P\| = 0\; \& \boldsymbol{Func}\; is\; not\; recursive\; \& \\ \quad F\; has\; no\; heap\; a\text{-}locs \quad //Strong\; Update \\ \quad SM'_e = \{\{SM_e - SM_e(v)\} \cup \\ \quad \{(v, SR(R1))\} \mid v \in F\} \\ else \quad //Weak\; Update \\ \quad SM'_e = \{\{SM_e - SM_e(y) \mid y \in \{F \cup P\}\} \cup \\ \quad \{(v, \nabla(SR(R1) \cup SM_e(v))) \mid v \in F\} \cup \\ \quad \{(p, \top) \mid p \in P\}\} \end{array}$
5. SSA Phi	$R_{n+1} := \phi(R_1, R_2, ..., R_n)$	$SR = \{SR - SR(R_{n+1})\} \cup \{R1, \nabla(\bigcup_{i \in (1,n)} SR(R_i))\}$

Table I: *Transfer functions for each instruction in a procedure Func. Here, s denotes the size of dereference in a memory access instruction.*

fully accessed `a-loc` (strong update), the existing symbolic values of the destination memory location is replaced by the symbolic set. Otherwise, the new symbolic values are unioned with the existing ones to obtain the symbolic value set of fully accessed `a-locs` (weak update). The partially accessed `a-locs` are assigned symbolic $\top$.

Memory regions corresponding to the stack frame of a recursive procedure or to heap allocations potentially represent more than one concrete `a-loc`. Hence, the assignments to their `a-locs` are also modeled by weak updates.

5. SSA Phi Function: $e : R_{n+1} = \phi(R_1, R_2, ..., R_n)$
At join points in the control-flow of a procedure, the symbolic value sets from all the predecessors are unioned to obtain a new symbolic value set.

5.3. Interprocedural propagation

Interprocedural analysis requires the correct handling of symbolic values at callsites and return points.

Several binary analysis frameworks [Zhang et al. 2007; Balakrishnan and Reps 2004], including SecondWrite [Anand et al. 2013a], implement various analyses to recognize the arguments. Once the arguments are recognized, formal arguments and returns are represented as a part of procedure definition and actual arguments and returns are explicitly represented as a part of a call instruction in the IR.

The symbolic value set of a formal argument for a procedure *P* is computed by unioning the symbolic value sets of corresponding actual arguments across all the call-sites for procedure *P*. Mathematically, the initialization of formal f_i of procedure *P*, where a_{ci} represents the corresponding actual argument at a callsite c, is represented as

$$SR = \{SR - SR(f_i)\} \cup \{(f_i, \nabla(\bigcup_{\forall c \in CallSites(P)} SR(a_{ci})))\} \tag{4}$$

The return variables are also handled in a similar manner. In order to propagate the symbolic values of `a-locs`, the memory symbolic maps from each call site need to be unioned to determine the symbolic map at entry point P_{entry} of a procedure P.

$$SM_{P_{entry}} = \bigcup_{\forall c \in CallSites(P)} SM_c \tag{5}$$

Similarly, the symbolic map just after a call instruction C, is computed by unioning the symbolic maps at all the return points in the called procedure *P*.

Externally called procedures are handled in one of the following three ways. First, procedures which are known not to affect the memory regions (e.g. `puts, sin`) are modeled as identity transformers (a NOP). External procedures like `malloc`, which create a memory region, are also modeled as identity transformers because these procedures are already handled by defining a memory abstraction `HeapRgn` corresponding to each allocation site. External procedures like `free`, which destroy a memory region, are conservatively modeled as NOP. Next, unsafe but known external procedures (e.g. memcpy) are handled by widening the symbolic value set of all `a-locs` in the memory regions possibly accessed by the procedure. Unknown external procedures (which include user defined libraries) are handled by widening the symbolic value set of registers and all `a-locs` in all the memory regions. This is an extremely rare scenario because most binary executable programs call external procedures from known libraries. We did not observer this scenario is our experiments. If a binary program calls a procedure from a user defined library, users can optionally provide information about memory regions accessed in such procedure to avoid an excessive loss of precision.

6. DEMAND-DRIVEN MECHANISM

In this section, we present `DemandSym` which enables the adaptation of `ExecSVA` to a particular client analysis. `DemandSym` computes the set of data-objects which are critical for a target analysis and ExecSVA is updated to compute abstraction for only this limited set. As described in Section 1, symbolic analysis can be employed for a variety of applications such as redundancy removal, data-dependence analysis and security analysis. DemandSym enables an efficient application of ExecSVA to any of these client analyses. In Section 6.3, we demonstrate an example of such an adaptation by extending our analysis for detecting information-flow violations.

6.1. Demand-driven Set

Below, we discuss our method of computing a limited set of data objects necessary to implement a particular client analysis. This required set of data objects (variables and memory locations) is represented as the Demand Set.

$$\texttt{Demand Set} = \begin{cases} \texttt{SR} : \textit{Set of required variables} \\ \texttt{SM} : \textit{Set of required } \texttt{a-locs} \end{cases}$$

Sets SR and SM are collectively referred to as Demand Set. We refer to an element of set Demand Set as a Demand Object.

Figure 8 presents the rules for computing Demand Set. We employ the logical inference form[2] for representing the deduction rules for computing the sets SR and SM. The rules are applicable to operations in the IR, but we present C-like pseudo instructions for ease of understanding. The rules constitute a backward analysis, where the instructions are traversed in a demand-driven backward dataflow order.

At the beginning of the analysis, SM is initialized as an empty set and SR is initialized with the data objects employed at the required program locations in the client program analysis, denoted as DemandInit. For example, in case of data-dependence analysis, set DemandInit comprises data-objects which need to be tested for dependence.

Given an initial set of elements in Demand Set, the rules presented in Fig. 8 analyze each program operation to update Demand Set accordingly. In case of an assignment operation, if the destination is already a Demand Object, the source operand is also added to the set. In case of arithmetic, logical and phi operations, the source operands are added to set SR, if it already contains the destination.

Memory operations employ Value Set Analysis (VSA) [Balakrishnan and Reps 2004] to update Demand Set. In case of a memory load operation, the a-locs present in the value set of the source operand are added to set SM only if the loaded value is already a Demand Object. Similarly, a value employed in a memory store operation is considered a Demand Object if any of the possibly accessed a-locs is an element of set SM.

Interprocedural rules in Fig. 8 depend on whether the called procedure is an internal or external procedure. The distinction is required because the procedure body of externally called procedures is not present inside the binary application being analyzed. In case of a call to an internal procedure, an actual argument value at the call site is added to SR if the corresponding formal argument is already present in SR. A return value also results in a similar update of SR. If the actual return value at the call-site is present in SR, then all the return variables in the procedure definition are also considered as Demand Objects.

A call to an external procedure is handled in one of the following two ways. If the prototype of the called procedure is available, then the call is modeled by adding all actual arguments and their underlying a-locs to Demand Set if the return value is a Demand Object. Otherwise, a call to a procedure with unknown prototype is modeled as a NOP to avoid an excessive loss of precision. As explained in Section 5.3, these scenarios are extremely rare because most binary executable programs call external

[2]The expression $\frac{\texttt{Premise \#1} \quad \texttt{Premise \#2} \ .. \ \texttt{Premise \#n}}{\texttt{Conclusion}}$ states that premises above the line allow us to derive the conclusion below the line.

Helper Variables

$\mathtt{VS(R)}$: Value Set of object $\mathtt{R}$
$\mathtt{R} \rightarrow \mathtt{z}$: $\mathtt{a\text{-}loc}$ $\mathtt{z} \in \mathtt{VS(R)}$
$\mathtt{OP}$: Arithmetic, Logical and Casting operators
$\mathtt{ARG_T}$: Set of parameters of procedure $\mathtt{T}$
$\mathtt{RET_T}$: Set of variables at actual return-sites in procedure $\mathtt{T}$
$\mathtt{FORM_i}$: Variable for $\mathtt{i}$th formal parameter of a procedure
$\mathtt{ACT_i}$: Variable for $\mathtt{i}$th actual parameter at a callsite
$\mathtt{F}$: An internal procedure
$\mathtt{X}$: An external procedure

Initialization

$\mathtt{SR} \leftarrow \mathtt{DemandInit}$; $\mathtt{SM} \leftarrow \{\ \}$

Rules

$$\mathtt{I{:}R1{=}R2}\ \frac{\mathtt{R1} \in \mathtt{SR}}{\mathtt{R2} \in \mathtt{SR}}$$

$$\mathtt{I{:}R1{=}R2\ OP\ R3} \left\{ \frac{\mathtt{R1} \in \mathtt{SR}}{\mathtt{R2} \in \mathtt{SR}} \quad \frac{\mathtt{R1} \in \mathtt{SR}}{\mathtt{R3} \in \mathtt{SR}} \right.$$

$$\mathtt{I{:}Rx{=}}\phi\mathtt{(R1,R2,..,Rn)} \left\{ \frac{\mathtt{Rx} \in \mathtt{SR}}{\mathtt{R1} \in \mathtt{SR}} \quad \frac{\mathtt{Rx} \in \mathtt{SR}}{\mathtt{R2} \in \mathtt{SR}} \quad \ldots \quad \frac{\mathtt{Rx} \in \mathtt{SR}}{\mathtt{Rn} \in \mathtt{SR}} \right.$$

$$\mathtt{I{:}R1{=}{*}R2} \quad \frac{\mathtt{R1} \in \mathtt{SR} \quad \mathtt{R2} \rightarrow \mathtt{z}}{\mathtt{z} \in \mathtt{SM}}$$

$$\mathtt{I{:}{*}R1{=}R2} \quad \frac{\mathtt{R1} \rightarrow \mathtt{z} \quad \mathtt{z} \in \mathtt{SM}}{\mathtt{R2} \in \mathtt{SR}}$$

$$\mathtt{I{:}\ R1{=}call\ F} \begin{cases} \forall_{\mathtt{i} \in \mathtt{ARG_F}} \dfrac{\mathtt{FORM_i} \in \mathtt{SR}}{\mathtt{ACT_i} \in \mathtt{SR}} \\ \forall_{\mathtt{i} \in \mathtt{RET_F}} \dfrac{\mathtt{R1} \in \mathtt{SR}}{\mathtt{i} \in \mathtt{SR}} \end{cases}$$

$$\mathtt{I{:}\ R1{=}call\ X} \begin{cases} \forall_{\mathtt{i} \in \mathtt{ARG_X}} \dfrac{\mathtt{R1} \in \mathtt{SR}}{\mathtt{ACT_i} \in \mathtt{SR}} \\ \forall_{\mathtt{i} \in \mathtt{ARG_X}} \dfrac{\mathtt{R1} \in \mathtt{SR} \quad \mathtt{ACT_i} \rightarrow \mathtt{z}}{\mathtt{z} \in \mathtt{SM}} \end{cases}$$

Fig. 8: *Deduction rules for computing `Demand Set`. Rules are presented in a logical inference form ($\frac{\mathit{Premise\ \#1}\quad \mathit{Premise\ \#2}\ ..\ \mathit{Premise\ \#n}}{\mathit{Conclusion}}$). These rules constitute a backward analysis, where a set of premises that are true after an instruction allow us to derive the conclusion that will hold true before the instruction.*

procedures from known libraries. We do not model side effects of external procedures to balance precision and utility by limiting computation of excessive demand objects.

The `Demand Set`, comprising `SR` and `SM`, captures all the variables and memory locations which can possibly impact the value of the elements in set `DemandInit`. This reduced set is employed to compute the symbolic abstraction in the program.

6.2. Demand-Driven Analysis

`DemandSym` computes an approximation of symbolic abstraction of each demand-driven data object at each program point. The analysis presented in Section 5 is modified to account for demand-driven computation. Specifically, the transfer functions presented in Table I are updated to compute the symbolic abstraction for only the data objects which are part of Demand Set. In Table I, transfer functions in row 1,2,3 and 5 compute the symbolic abstraction for program variables while transfer function in row 4 updates the symbolic abstraction for memory locations. The set of transfer functions in Table I, `TF` at a program location `e`, can be represented as follows:

$$TF_e = \begin{cases} \texttt{RegT(r)} : \texttt{Transfer functions for r (Row 1,2,3 and 5 in Table I)} \\ \texttt{MemT(M)} : \texttt{Transfer functions for a-locs} \in \texttt{M(Row 2 5 in Table I)} \end{cases}$$

The above set `Transfer Functions` is updated as follows to reflect demand-driven computation.

$$TF'_e = \begin{cases} \texttt{RegT'(r)} : \texttt{if r} \in \texttt{SR then RegT(r) else } \varnothing \\ \texttt{MemT'(M)} : \texttt{if M} \in SM_e \texttt{ then MemT(M) else } \varnothing \end{cases}$$

Effectively, `DemandSym` can be represented through the following combination of inputs and outputs

`Input:` `DemandInit`
`Output:` { `SR(r)` $\forall$ `r` $\in$ `Input`}

6.3. Example: Information-flow policy enforcement

Next, we employ `DemandSym` to adapt `ExecSVA` for detecting information-flow violations. We present information flow vulnerability detection as a case study for our `DemandSym` framework and demonstrated that it is very effective in improving the scalability. `DemandSym` framework is not applicable to buffer overflow, heap spray and other kinds of vulnerabilities that cannot be easily represented as information-flow vulnerabilities.

Several information-flow tracking systems express a policy using the concept of labels [Chang et al. 2008]. There are three dimensions that characterize a policy: label description, label initializations and label checks. *Label description*, `LB`, specifies the underlying labels; *Label initializations*, `LBInit`, correspond to the program sites that introduce labels into the program; while *Label checks*, `LBCheck`, denote the sensitive program locations where an information-flow violation might arise if an untrusted label reaches such locations. These can be represented as follows

`LB`: { `Set of labels` }
`LBInit`: { `(x,l), l` $\in$ `LB` }
`LBCheck`: { `y` }

In addition, these frameworks also define a function Ω to combine the labels at a program location. For example, if the set `LB` contains labels `tainted` and `untainted`, Ω function to combine these two labels can be defined as follows.

Ω(`tainted,untainted`) = `tainted`.

A more expanded definition is required for Ω function in case of multiple labels [Chang et al. 2008].

The demand framework provides a generic and programmable application programming framework for specifying such kinds of information-flow policies. The above representation can be mapped to Demand framework as follows:

Input: { `LBCheck` }
Output: { `SR(y)` $\forall$ `y` $\in$ `Input`}

Next, we define a function, `LBMap`, to determine the labels at required programs locations (`LBCheck`) using `DemandSym` output. Suppose `IRSym(S)` is the set of `IR symbols` present in a symbolic value set `S`. As presented in Fig 7, `IR symbols` constitute symbolic alphabets in our grammar.

$$\texttt{LBMap(y)} : \Omega \; \{ \; \texttt{LBInit(r)} \; \forall \; \texttt{r} \in \texttt{IRSym(SR(y))} \; \}$$

The presence of unsafe label in LBMap(y) signifies a vulnerability. Below, we provide an example of policy specification using format-string vulnerability.

Format string flaws arise due to an unsafe implementation of variable-argument procedures in C library. In case of a variable-argument procedure like `printf`, a `format string` argument specifies the number and type of other arguments. However, there is no runtime routine to verify that the procedure was actually called with the arguments specified by the `format string`. As detailed in [Cowan and et al. 2001], an attacker can corrupt the format string and thereby take control of the program by modifying relevant memory locations.

In order to expose a format string vulnerability, a tool needs to detect the flow of information from an untrusted source to the `format string` argument of a variable argument procedure. Hence, this policy is specified as follows

LB: { `tainted, untainted` }
LBInit: { `Callsites corresponding to external input procedures` }
LBCheck: { `Format String Arguments` }

Based on above description, the presence of `tainted` label in LBMap(y) for any `y` $\in$ `LBCheck` signifies a vulnerability.

7. DISCUSSIONS

In this section, we discuss some of the limitations and assumptions behind our techniques. We first discuss the assumptions related to underlying SecondWrite binary analysis framework followed by specific limitations related to our techniques.

7.1. SecondWrite Assumptions

As we described in Section 3, SecondWrite translates the input x86 binary code to a workable program represented in the intermediate representation (IR) of the LLVM Compiler [Lattner and Adve 2004]. SecondWrite is based on a conventional wisdom that static analysis of executables is a very difficult problem and statically handling every program may be an elusive goal. Hence, its goal is to expand the static envelope based on a set of assumptions. Below, we briefly discuss the assumptions behind SecondWrite, Anand et al [Anand et al. 2013a], Smithson et al. [Smithson et al. 2013] and Wazeer et al. [ElWazeer et al. 2013a] discuss all the assumptions in more detail.

→ **Disassembly assumptions**: As we described briefly in Section 3, *binary characterization* employed by the underlying disassembler in SecondWrite derives possible addresses using an assumption that an indirect control transfer instruction requires an absolute address operand [Anand et al. 2013a; Smithson et al. 2013]. This assumption is essential to overcome the general undecidability of program disassembly. A compiled code is expected to adhere to this convention unless it has been generated to be position independent. A position independent code computes the indirect control transfer targets based on the address at which the program is loaded. Hence, binary characterization fails to identify these address because these addresses are not present in binary image. Smithson et al. [Smithson et al. 2013] discuss this assumption and possible resolution in more detail. SecondWrite does not yet support advanced x86 features such as SSE and other advanced instructions. Benchmarks that contain such instructions have not been considered in our analyses.

→ **Procedure Boundary and Control-flow assumptions**: The above method based on binary characterization is not sufficient for discovering indirect branch targets where addresses are calculated in binary. Hence, various procedure boundary determination techniques, such as ending the boundary at beginning of next procedure, are also proposed [Smithson et al. 2013] to limit possible targets. SecondWrite also implements several additional techniques [ElWazeer et al. 2013a] to recover procedure boundaries and eliminate spurious procedures from recovered IR. These methods are based on assumptions such as procedures should have a prologue and an application code should not contain certain code semantics such as memory access to an address outside memory image. Wazeer et al. [ElWazeer et al. 2013a] discuss such assumptions in more detail.

→ **Self Modifying code**: Like most static binary tools, SecondWrite does not handle self modifying code. Various tools [Wang et al. 2008; Thakur et al. 2010] statically detect the presence of self-modifying code in a program. Such a tool can be integrated in our front-end to warn the user and to discontinue further operation.

→ **Obfuscated Code**: We have not tested our techniques against executables with hand-coded assembly or containing techniques that thwart static disassembly by obfuscating the discovery of control flow [Linn and Debray 2003].

7.2. Assumptions in symbolic analysis framework

Below, we discuss some of the assumptions behind the methods presented in this paper.

→ **Architectural assumptions**: Out of the three techniques presented in the paper, `ExecSVA` and `DemandSym` are generic and are applicable to any ISA. The remaining technique, `MemRecovery`, contains several rules that currently limits its applicability to only x86 ISA. Extending `MemRecovery` to other ISAs will be explored in future work.

→ **Memory assumptions**: Similar to most executable analysis frameworks [Balakrishnan and Reps 2004; 2007; Schwarz et al. 2001], our techniques assume that executables follow the *standard compilation model* where each procedure may allocate an optional stack frame in only one direction and each variable resides at a fixed offset in its corresponding region. We also assume that in x86 programs, a particular register `esp` refers to the top of memory stack. This assumption is expected to hold in all practical scenarios because x86 ISA inherently makes this assumption. For example, x86 `call` instruction moves value stored in instruction pointer, `eip`, to `esp`. Similarly, `return` instruction directly modifies `esp`. An assembly code not adhering to this convention would be extremely hard to write.

→ **Library code**: Since our tool relies on discovery of heap allocation using `malloc` and specification of information points using external library functions, it does not support programs that statically link the standard C library. All other libraries can be statically linked in the program.
The dependence of our tool on identification of external library function names such as `malloc` does not contradict our claim about not requiring symbol table information in an executable. The information regarding library and procedures that are

linked dynamically to an executable is always present in its dynamic symbol table. Such information cannot be stripped from an executable, otherwise it will fail to load and execute properly. On the other hand, a symbol table in an executable containing information about functions within an executable is not required for proper functioning of an executable and is usually not present in an off-the-shelf executable. Our techniques are applicable to such off-the-shelf executables that do not contain such symbol table information.

→ **Custom Heap Management**: The presence of custom heap management in an application results in unsoundness in our analysis. An allocation site inside such an application appears as a call to a custom procedure (such as `tcmalloc` [TCMalloc]) as opposed to a call to standard `malloc` library procedure. Our analysis does not detect such allocation sites and fails to propagate symbols across these memory locations.
Users can optionally eliminate the above unsoundness by specifically identifying the custom allocation procedure. Memory allocated at all corresponding allocation sites can then be considered as a single memory `a-loc` in our analysis. This will still results in an imprecise analysis but will prevent the unsoundness.

→ **Information-flow Analysis**: As presented in Section 6, `DemandSym` enables the adaptation of `ExecSVA` to a particular client analysis. `ExecSVA` might end up computing an abstract value of $\top$ for some data objects. `DemandSym` treats such values in an unsound manner by assuming that they do not contain any value. This choice is governed by a need to balance the precision and overhead while doing static program analysis and to avoid excessive false positives.
In its current form, `DemandSym` framework can only be applied for vulnerabilities that can be directly represented as information-flow violations. Buffer overflow, heap overflow and other kinds of security vulnerabilities cannot be easily represented as information-flow vulnerabilities. Extending our framework for other kinds of vulnerabilities will require extra research effort and we expect that our work would be used as a stepping stone towards that research effort.
Several applications implement input validations to verify and sanitize the inputs from untrusted sources prior to its use at a sensitive sink [Balzarotti et al. 2008]. `DemandSym` framework currently fails to detect the presence of such sanitization mechanisms. This can adversely produce a few false positives in such applications. In future work, we plan to modify `DemandSym` framework to accurately model sanitization mechanisms.

8. RESULTS

Our techniques are implemented as part of the SecondWrite framework presented in Section 3. The evaluation is performed on benchmarks from the SPEC2006 and OMP2001 suites and some real world programs, as listed in Table II. All the benchmarks of SPEC2006 suite and OMP2001 suites that have been shown to be supported by SecondWrite framework [Anand et al. 2013a] are included in this list. The following benchmarks from SPEC2006 suite - gamess, gems, tonto and wrf, are not included in this because these benchmarks contain some x86 instructions that are not supported in SecondWrite framework. Benchmarks are compiled with gcc v4.3.1 with O3 flags (full optimization) and results are obtained on a 2.4GHz 8-core Intel Nehalem machine running Ubuntu 12.04. In all figures, AVG refers to arithmetic mean, AVG-C refers to arithmetic mean for C benchmarks and AVG-C++ refers to arithmetic mean for C++ benchmarks.

8.1. Workable Representation and Precise Memory Model

Figure 9 and Fig. 10 present the statistics regarding `MemRecovery` for obtaining precise memory model and workable IR. We only present statistics for benchmarks containing

Application	Source	Lang	LOC	# Proc	Time(s)	Mem (MB)
bwaves	Spec2006	Fortran	715	22	4.25	24.47
lbm	Spec2006	C	939	30	0.8	1.03
equake	OMP2001	C	1607	25	0.64	3.62
mcf	Spec2006	C	1695	36	0.31	2.85
art	OMP2001	C	1914	32	0.36	2.74
wupwise	OMP2001	Fortran	2468	43	1.37	5.68
libquantum	Spec2006	C	2743	73	1.30	6.30
leslie3d	Spec2006	Fortran	3024	32	8.24	23.72
namd	Spec2006	C++	4077	193	19.46	111.53
astar	Spec2006	C++	4377	111	1.49	8.39
bzip2	Spec2006	C	5896	51	4.8	90.27
milc	Spec2006	C	9784	172	41.16	19.68
sjeng	Spec2006	C	10628	121	9.93	34.98
sphinx	Spec2006	C	13683	210	7.11	31.19
zeusmp	Spec2006	Fortran	19068	68	37.85	285.48
omnetpp	Spec2006	C++	20393	3980	21.66	58.24
hmmer	Spec2006	C	20973	242	12.13	36.52
soplex	Spec2006	C++	28592	1523	21.21	144.14
h264	Spec2006	C	36495	462	29.56	220.53
cactus	Spec2006	C	60452	962	25.65	185.05
gromacs	Spec2006	C/Fortran	65182	674	47.82	252.33
dealII	Spec2006	C++	96382	15619	114.30	240.18
calculix	Spec2006	C/Fortran	105683	771	192.99	404.32
povray	Spec2006	C++	108339	3678	71.01	242.61
perlbench	Spec2006	C	126367	2183	94.18	210.37
gobmk	Spec2006	C	157883	4188	60.66	242.19
gcc	Spec2006	C	236269	6426	663.69	490.68
xalan	Spec2006	C++	267318	30062	464.97	183.75
gzip	Compress	C	10671	98	1.42	20.06
tar	Compress	C	20518	343	9.58	18.85
ssh	Web clinet	C	73335	887	40.57	22.55
lynx	Browser	C	135876	2106	140.08	73.01
apache	WebServer	C	232931	2026	37.98	232.12
MySQL	Database Server	C++	1734500	11932	3745	3242.61

Table II: *Applications Table. Time and Memory columns show the analysis time and storage requirements for* `ExecSVA`.

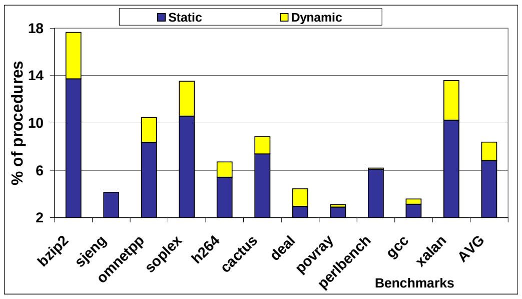


Fig. 9: *Percentage of procedures with unknown CTIs. The static represents cases when constraint solvers succeed.*

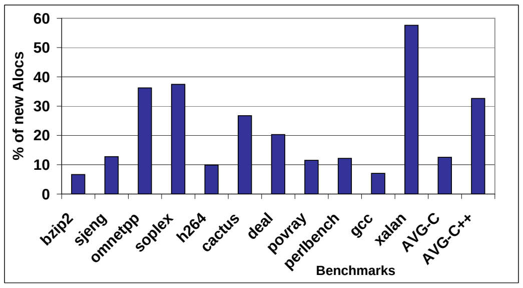


Fig. 10: *Additional alocs added as a result of constraint solvers, normalized to original number of alocs.*

non-negligible *Unknown CTIs* because our techniques are meant for such challenging executables. Here, negligible is defined as # of *Unknown CTIs* ≤ 10 or # of procedures containing *Unknown CTIs* $\leq 1\%$). Of the 33 programs in Table II, 11 had non-negligible unknown CTIs, including all five largest benchmarks out of total seven C++ benchmarks. This is not surprising because C++ benchmarks tend to contain more unknown CTIs because of the presence of virtual function calls.

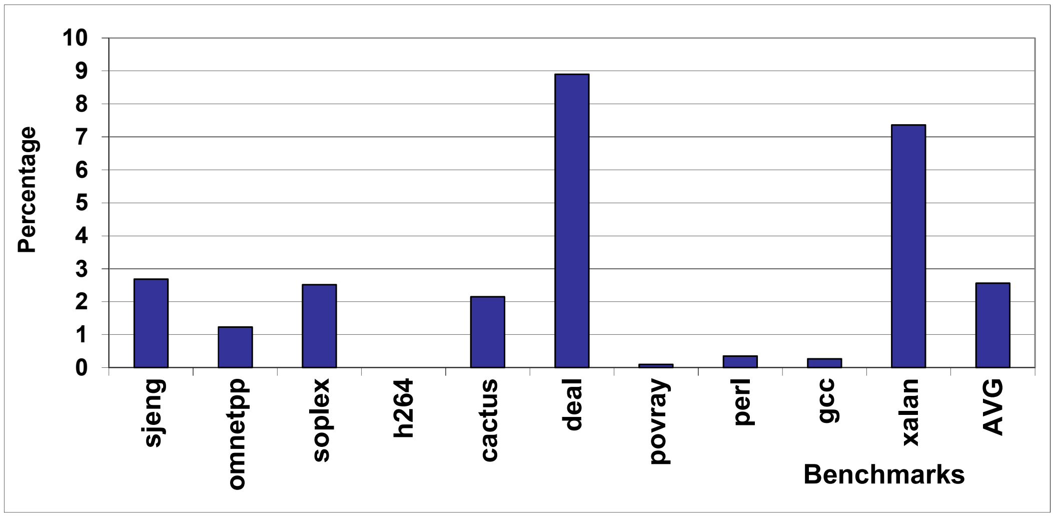


Fig. 11: *Percentage of non zero indirect CTIs among all indirect CTIs.*

Application	Num of Indirect CTI	Num of Indirect CTI with non-zero `StackDiff`
soplex	527	4
deal	1016	108
xalan	10079	289

Fig. 12: *Number of Indirect CTI with non-zero* `StackDiff`.

Figure 9 presents the fraction of procedures containing *Unknown CTI* in each of these benchmarks. It divides this fraction into scenarios where the static mechanism was able to determine the value of `StackDiff` and where the dynamic mechanism was required to maintain the workability. Case 1 (Section 4.1) does not arise because we employ the prototypes for standard library procedures. We never hit the invalidation conditions stipulated in Fig. 5, justifying our assumptions.

Figure 10 illustrates the additional a-locs derived as a result of successful constraint solutions, normalized with respect to original a-locs of type `Stack` (Section 5.2). As evident, we were able to obtain 10% more `a-locs` in C benchmarks and 30% more `a-locs` in C++ benchmarks on average. This enhanced `a-locs` abstraction is employed in our symbolic value analysis framework.

Further analysis of our results in Fig. 9 elucidates that we never come across Case 3 during our experiments. The analysis of source-code reveals that the value of `StackDiff` at each indirect CTI is constant. However, an imperfect alias analysis of indirect CTI targets results in defining a larger set of targets, many of which have different `StackDiff` values. Figure 11 lists the fraction of procedures in each benchmark in Fig. 9 that have non-zero `StackDiff` values. An imperfect alias analysis results in gathering up of several of such procedures in target call set of indirect CTI in each benchmark, resulting in an unknown CTI.

Table 12 further lists the number of indirect CTI in the benchmarks in Fig. 9 that are confirmed to have a non-zero `StackDiff` value. Indirect CTI in rest of the benchmarks have zero `StackDiff` value. This result reveals that even though the fraction of indirect CTI with non-zero `StackDiff` value is small, a simple method of always assuming a

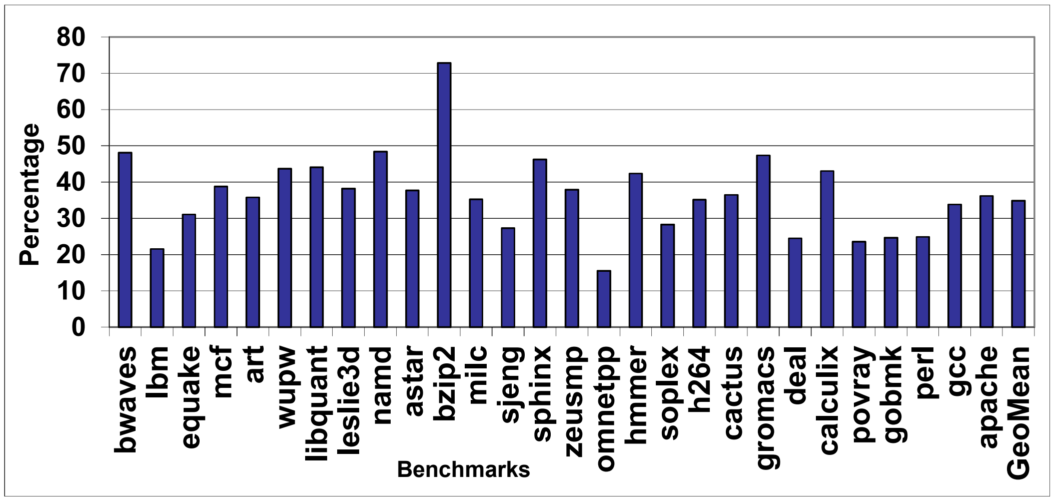


Fig. 13: *Fraction of symbolic expressions containing symbolic alphabets propagated through memory locations.*

zero value of `StackDiff` can never guarantee to obtain a workable IR from binary executable code.

In order to understand the source-code pattern that results in a non-zero CTI, we analyzed a callsite in one of the benchmark from Table 12. In `soplex`, an unknown indirect CTI arises due to call to a virtual procedure with following prototype:

```
virtual SpxID selectEnter()
```

If a procedure returns a structure, `gcc` creates a binary code with non-zero `StackDiff`. This is one such scenario that results in a non-zero `StackDiff` in spite of a standard `cdecl` calling convention [SourceForge 2013]. In more complex programs with advanced calling conventions, unknown CTIs might arise at a higher frequency.

8.2. ExecSVA

Table II shows the analysis time and storage requirements of `ExecSVA` on various applications. The numerical value of `Limit`, the maximum size of a symbolic value set, was kept to `5`. The analysis time and the required storage is largely a function of the number of procedures in the benchmark. The analysis scales well to mid-size programs. But the time increases for large benchmarks. For example, the analysis time for mySQL is more than an hour.

In order to understand the importance of tracking memory locations, we obtain the percentage of symbolic expressions that containing at least one symbolic alphabet propagated through a memory location, as a percentage of symbolic expressions for all IR variables. Fig. 13 elucidates an interesting characteristic of the obtained symbolic expressions. We observe that 35% of symbolic expressions contain alphabets propagated through memory locations. In absence of an abstraction for memory locations, the analysis would have introduced a new alphabet in all these expressions. This validates our central contribution that tracking memory locations is essential for effective symbolic analysis on executables.

In order to understand our symbolic abstraction, we divided the objects into various categories according to the size of their symbolic value set as shown in Fig. 14. On

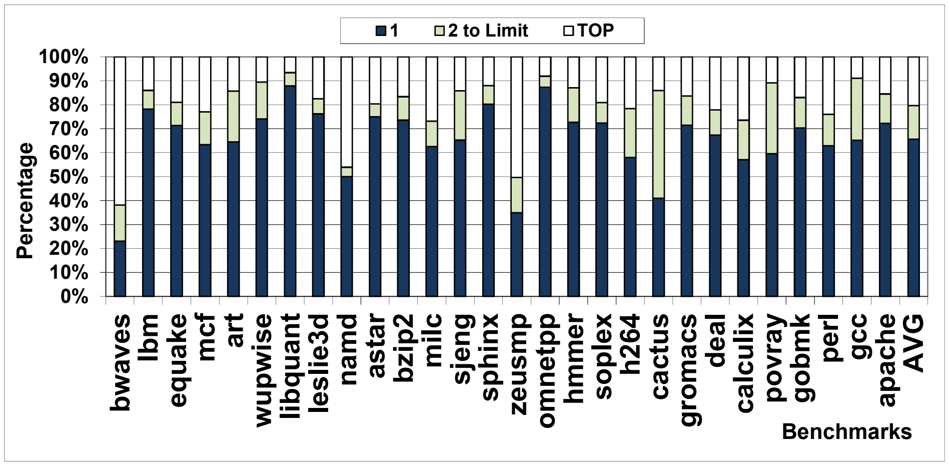


Fig. 14: *Symbolic Value Set Visualization.*

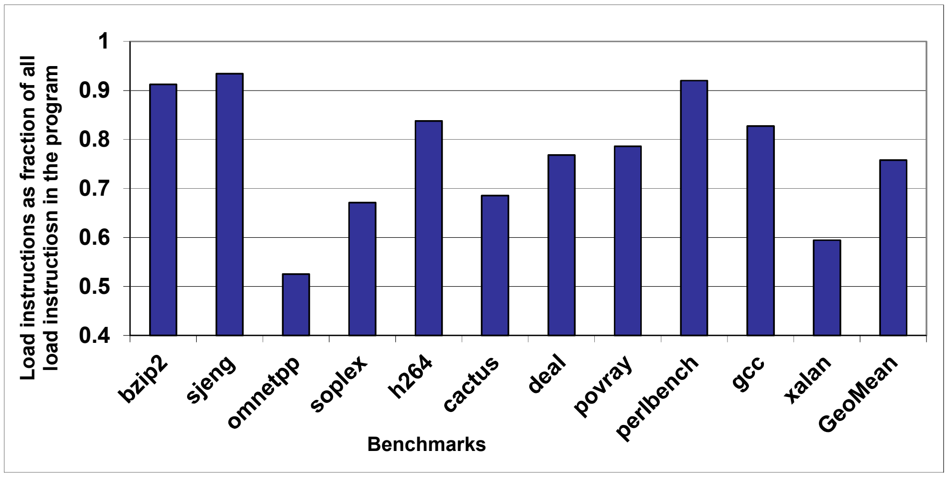


Fig. 15: *Variables requiring a new symbolic alphabet in presence of additional a-locs.*

average, 64% of objects can be abstracted with a single symbolic expression, 16% of objects need multiple expressions and 20% of objects cannot be represented with finite symbolic abstraction ($\top$). Maintaining a symbolic value set instead of a single symbolic expression allows us to maintain this extra precision for 16% of data objects.

Figure 15 captures the enhancement in the precision of `ExecSVA` with the presence of additional `a-locs` derived by the `MemRecovery` mechanism. According to Table I, a load instruction accessing an unknown memory location is represented by a new symbolic alphabet. Figure 15 demonstrates the decrease in the number of load instructions requiring a new alphabet while employing additional *a-locs*. The presence of additional *a-locs* enhances the precision of symbolic value analysis by 10% to 50% in several programs.

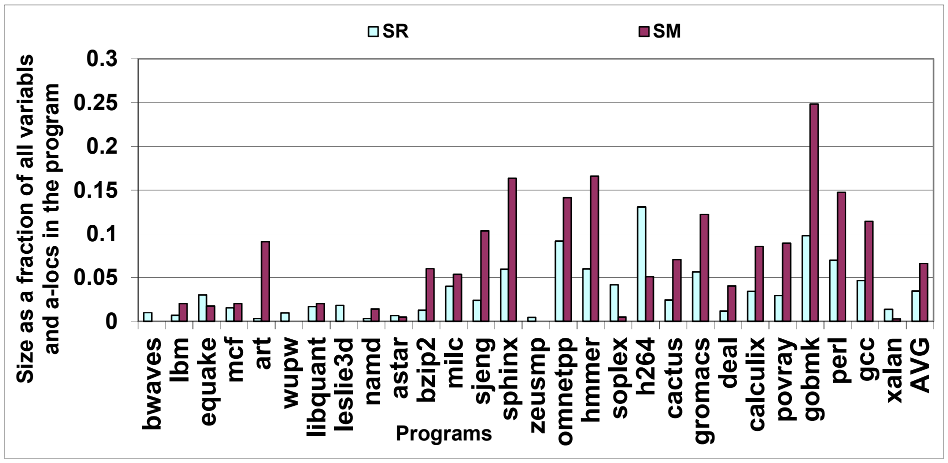


Fig. 16: *Size of* `Demand Set` *(*`SR` *and* `SM`*) normalized (=1.0) to all* `variables` *and* `a-locs` *respetively.*

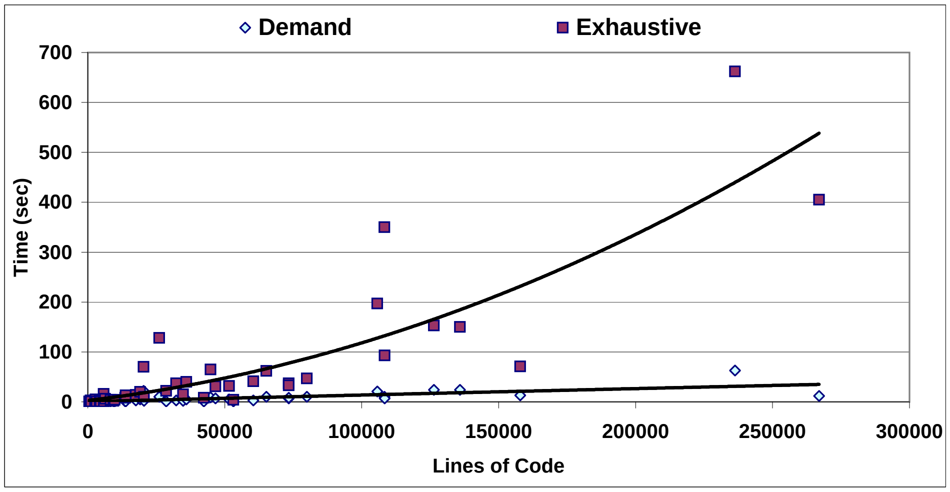


Fig. 17: *Scalability of demand driven and exhaustive analysis with increasing lines of code. Trendlines capture the scalability of both versions of analysis.*

8.3. Demand-driven Analysis

Recall from Section 2, `DemandSym` enhances the scalability of `ExecSVA`. Here, we quantify this enhancement.

Figure 16 presents the size of `Demand Set`, `SR` and `SM` for detecting format string vulnerability. The sizes of `SR` and `SM` are normalized against the total number of variables and `a-locs` in the program respectively. Figure 16 shows that these rules are highly efficient in decreasing the overall analysis requirement. This enables `DemandSym` to only analyze around 5-10% of total objects, on average, without sacrificing the precision. C++ and fortran programs do not have many `format string` calls. Hence, they have relatively small `SR` and `SM` set.

Application	LOC	Vulnerability	Type
mingetty 1.08	500	-	-
csplit 8.17	1060	-	Format String
muh 2.05c	2857	CVE-2000-0857	Format String
pfingerd 0.7.8	4689	NISR16122002B	Format String
gzip 1.2.4	5830	CVE-2005-1228	Directory Trav.
ez-ipup3.0.10	6335	CVE-2004-0980	Format String
gif2png 2.5.2	9354	CVE-2010-4695	Directory Trav.
wu-ftpd 2.6.0	17576	CVE-2000-0573	Format String
tar 1.13.19	20518	CVE-2001-1267	Directory Trav.
KeePassX0.4.3	26089	-	-
yafc 1.1.1	32241	**NEW**	Directory Trav.
tnftp 2010	34762	-	-
gftp 2.0.19	42390	-	-
irc2 2011	44837	**NEW**	Directory Trav.
wget 1.13	46611	-	-
sudo 1.8	53144	CVE-2012-0809	Format String
openssh 6.0p	73335	-	-
ayttm 0.6.3	80013	**NEW : CVE-2015-6930.**	Format String
curl 7.30.0	122248	**NEW**	Directory Trav.
BitchX 1.1	133728	**NEW**	Format String
lynx 2.8.7	135876	-	-
apache 2.2.17	232778	-	-
MySQL 5.6.11	1741774	-	-

Fig. 18: *Vulnerabilities discovered in real-world programs.*

Figure 17 highlights the ensuing enhancement in the scalability of `ExecSVA` as a result of `DemandSym`. It plots the variation in the time taken to analyze the programs using `DemandSym` with increasing lines of code and compares it with an exhaustive analysis. Figure 17 includes the programs listed in Fig. 18 as well as programs from complete *SPEC2006* benchmark suite. As evident, demand-driven analysis is approximately 10x more scalable than the exhaustive analysis. For example, the time to analyze *gcc*, a large SPEC2006 benchmark with 250,000 lines of code, reduces to less than a minute as compared to more than 11 minutes in exhaustive analysis. This scalability becomes more evident in programs like *MySQL* where demand mechanism was able to finish the analysis in 7 minutes (not shown in the graph) as compared to more than an hour of exhaustive analysis.

8.4. Application: Security Analysis

In this section, we discuss `DemandSym`'s ability to uncover standard vulnerabilities such as format string and directory-traversal attacks. A format string vulnerability was described in Section 6.3. A directory-traversal vulnerability typically arises when a filename supplied by an user is employed in a file-access procedure without sufficient validation. A user can include a .. (*dot dot*) within the response which might not be validated in some programs. In such scenarios, users will be able to gain an ability to ascend outside the authorized directory. This vulnerability can be uncovered in a similar manner, by assigning a *tainted* label to the inputs coming from an untrusted channel and raising an alarm at any use of a *tainted* value as a filename argument.

As explained in Section 1, analysis of vulnerabilities in binary executable code has several advantages over a source-code based analysis system. It enables a more rapid response to cyberattacks and enable users to certify the severity of attacks.

We evaluate DemandSym on a set of real-world programs listed in Fig. 18. These constitute widely deployed applications whose integrity is critical for functionality of a system. Figure 18 shows that DemandSym uncovers five previously unknown vulnerabilities, apart from detecting all previously known vulnerabilities in this set of programs. Next, we discuss the characteristics of these zero-day vulnerabilities.

ayttm: *ayttm* is vulnerable to a previously unknown format string attack. In *ayttm*, a procedure http_connect populates a variable inputline by receiving data from network using a call to external procedure recv. Then, inputline is assigned to a variable debug_buff using snprintf, which is further used as a format string argument in a printf call. This vulnerability has been confirmed by the developers.

BitchX: DemandSym exposes a format string vulnerability in *napster* plugin in BitchX. The behavior is similar to the vulnerability in *csplit*, where an input argument value is employed as a format string argument in a call to vsnprintf.

yafc, irc2, curl: DemandSym exposes directory-traversal vulnerabilities in each of these programs. These programs employ getenv to derive the name of the current directory and prepend the resulting value to derive the name of a file which is employed to open a file using a fopen call without any sanitization. As per several existing attacks [CVE 2013], an attacker might corrupt the environment variables, rendering the application susceptible to directory-traversal attacks.

These potential vulnerabilities in file transfer and internet relay clients can lead to security problems in atleast two scenarios. First, the client can be used as part of a web application that takes a form input and uses it to do an FTP transfer. Second, the client can be used in a setuid [Setuid] shell script or can be otherwise invoked from a setuid or setgid program. Setuid and setgid are Unix access rights flags that allow users to run an executable with the permissions of the executable's owner or group respectively. In this case, the attacker can use it to access files to which they otherwise don't have an access.

According to Mitre [CVE-MITRE 2015] and the original developers of these file transfer and internet relay chat applications, a web-application that employs the above applications should itself implement a sanitization behavior. Hence, these scenarios cannot be considered as vulnerabilities in these applications. Nonethless, these results establish the applicability of our tool in identifying information flows in a system that could be part of an attack. Thereater, system developers can identify the ideal location for implementing a sanitization check in the information flow path.

A more careful analysis of some of the potential vulnerabilities detected through our framework establishes the importance of precise memory analysis and also underscores the requirement of sanitization facility in future to further reduce false positives. Analysis of csplit demonstrates both these aspects. csplit is a well-known GNU *Coreutil* program. DemandSym detected a possible format string vulnerability in this utility. *csplit* declares a global variable suffix, which is initialized in procedure main using an input argument (optarg). Next, suffix is employed directly as a format string argument in a call to sprintf. DemandFlow flagged this unsafe flow from an external source to a format string argument. On careful analysis, we observed that global variable suffix is memory-allocated in the executable. DemandSym would not have uncovered this unsafe flow if the information is not propagated across memory locations. This example underscores the importance of our precise memory analysis for exposing information-flow vulnerabilities in executables. We notified the Coreutil developers about this vulnerability. They pointed to a sanitization procedure inside their code which was missed by our analysis. This observation demonstrates that the

```
static char * suffix ;

main:
......:
switch (..){
     case 'b':
     //Unsafe Initialization
       suffix = optarg;
       break;}

make_filename:
.......
sprintf(filename_space,
    suffix,
   //Format string Arg
    num);
```

(a) Source code snippet

```
0x8056160:  Fixed location for optarg;
0x80561ac: Memory address of suffix

(Address)        (Instruction)
main:
.......
804afb4:       mov    0x8056160,%eax  //Load from optarg
804afb9:       mov    %eax,0x80561ac  //Store to suffix
804afbe:       jmp    804b0b6 <main+0x1f8>

make_filename:
804a18e:      mov    0x80561ac,%eax    //Load from suffix
.......
804a1c6:      mov    %eax,0x4(%esp)    //Initialize format arg
804a1ca:      mov    %edx,(%esp)
804a1cd:      call   8048ec0 <sprintf@plt>
```

(b) Executable code snippet

Fig. 19: *Code snipped from* csplit *showing the format string vulnerability. Second operand is the destination in executable code.*

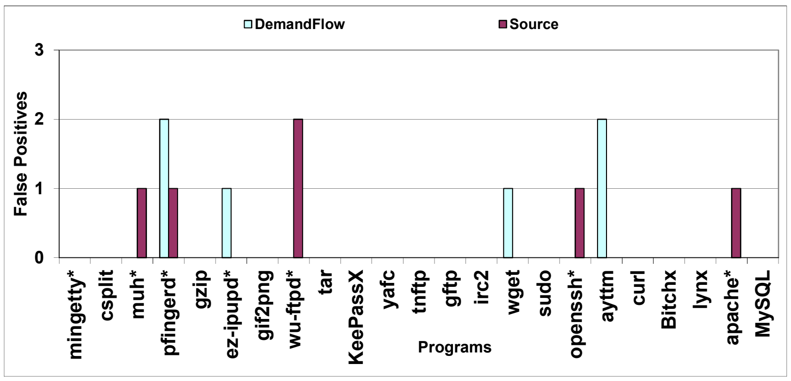


Fig. 20: *Format string vulnerability detection. * represent the programs where false positives are available from existing source-code tools.*

precision of our analysis can further be improved by including sanitization mechanism in our analysis.

Next, we establish the importance of reasoning about memory accesses for vulnerability detection. In order to simulate the functionality of previous tools [Cova et al. 2006], which do not track memory locations, the analysis presented in Section 6.2 is modified to compute the abstraction for only the variables. This is accomplished by disabling the rules in Table I for memory access instructions and by computing only IR. *The resulting analysis fails to unmask even a single vulnerability in the programs listed in Fig. 18*. This demonstrates the importance of a precise memory analysis for implementing a robust information-flow mechanism in executables.

False Positives: Figure 20 presents the false positives reported by DemandSym for the programs in Fig. 18 while detecting format string vulnerabilities, comparing the resulting statistics with the false positive reports generated by existing source-level static analysis tools (Oink [Oink 2015], CQual [Shankar et al. 2001] and others [Guyer

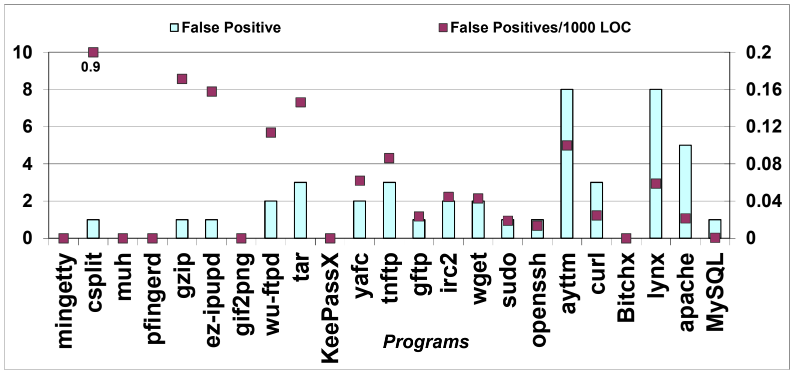


Fig. 21: *Directory traversal attacks.*

and Lin 2003])[3]. The Source bar in Fig. 20 represents the fewest false positive reported among the above tools. `DemandSym` reports similar false alarms as existing source-level tools for the programs listed in Fig. 18.

Figure 21 presents the corresponding statistics obtained for directory-traversal vulnerability. Even though `DemandSym` reports eight false positives for *lynx* and *ayttm*, it translates to less than 0.1 false alarms per 1000 lines of code. To the best of our knowledge, no existing source-level static analysis tool has reported directory-traversal statistics for the above set of programs; hence, the results could not be compared.

We further analyzed *lynx* and *ayttm* programs to understand the source of false positives while analyzing directory-traversal vulnerabilities. Out of eight false positives reported in *lynx*, atleast two false positives arose due to lack of considering sanitization function within our analysis. These two false positives were arising in *LyUpload* procedure. However, *LyUpload* does contain code to report a warning if .. is detected inside file name. Rest of the false positives seem to be due to imprecision in the analysis. In case of *ayttm*, we did not find any sanitization procedure that checked for presence of .. inside file name arguments. Hence, false positives in *ayttm* are most likely due to imprecision of analysis.

The false positive rate (FP/Total Reports) is 79.1% for above programs which is equivalent to 84% false positive rate [Chang et al.] reported by source-tools such as Oink and CQual [Oink 2015; Shankar et al. 2001].

Figure 16 shows that demand analysis enables us to narrow down our analysis to only 5% of original 5 million lines of code in these programs. Effectively, `DemandSym` only reports around 50 false positives from 250K lines of code. Corresponding statistics for SPEC benchmarks are presented in an expanded version of this paper [Anand et al. 2013b].

8.5. Application: Automatic Parallelization

Next, we substantiate the impact of symbolic analysis on dependence tests for program parallelization. The notion of data dependence captures the most important properties of a program for efficient parallel execution on multicores and parallel machines. The dependence structure of a program defines the necessary constraints on the order of execution of program components. Various dependence analyses such as array subscript analysis [Wolfe 1990], distance vectors [Banerjee 1979], ZIV tests (Zero induc-

[3]The programs with no corresponding results by source-tools are conservatively assumed to have zero false positives.

App	# Tests	#Success (Without mem based ExecSVA)	#Success (With mem based ExecSVA)	% Imp
2mm	18	14	18	28.5
3mm	26	20	26	30.0
atax	6	3	4	33.3
bicg	13	6	10	66.7
covar	19	16	19	18.7
doitgen	25	10	23	130.0
gemm	14	12	14	16.67
gemver	26	22	26	18.18
gesummv	13	9	12	33.3
jacobi	27	13	13	0
ft	127	37	43	16.2
lu	4866	1438	2078	44.5
bt	2866	1844	2237	21.3
sp	3317	2287	2815	23.1
AVG				34.3

Fig. 22: *Parallelization benchmarks (Polybench and NAS).*

tion variable), SIV tests (Single induction variable), and MIV tests (Multiple induction variables) [Goff et al. 1991] have been suggested for determining the dependence structure of a program and for determining parallel tasks.

These data dependence tests are more effective if the array subscript expressions are represented as affine expressions directly in terms of loop indices and loop invariants, rather than indirectly via other locations. As discussed in various parallelizing compilers such as Parafrase [Haghighat and Polychronopoulos 1996] and SUIF [Hall et al. 2005], a large percentage of parallelization benchmarks have array references with symbolic terms other than loop induction variables and have symbolic loop bounds.

Symbolic analysis has been suggested as an important technique for improving the data dependence decisions taken by a compiler in such scenarios. It is a very effective technique which represents array subscripts and loop bounds as a symbolic expression, describing its value in terms of constants, loop-invariant symbolic constants and loop indices. Standard dependence tests can then be employed to resolve data dependence queries [Hall et al. 2005; Blume and Eigenmann 1994].

ExecSVA, as presented in this paper, maintains symbolic abstractions for underlying memory locations thereby enabling the discovery of such affine expressions from executables also. Existing source-level symbolic frameworks [Haghighat and Polychronopoulos 1996] obtain these recurrence relations for only the variables while our ExecSVA analysis will obtain this recurrence relations for IR variables as well as for *a-locs*. Since a loop-index variable might be allocated to a memory location in an executable, ExecSVA recognizes recurrence expressions for memory-allocated loop index variables also, which cannot be recognized by applying existing source-level frameworks directly to executables.

Existing parallelizing compilers employ recurrence relations and affine expressions between array indices to characterize the dependence structure in two aspects. First, they try to disprove the loop carried dependence between pairs of subscripted references to the same array variable. Second, if dependence exists, they try to characterize the dependence by determining the actual distance in terms of number of loop iterations (referred to as distance vector) between two accesses to the same memory location. The tests are considered to be successful when a precise answer can be obtained for any of the above.

We have tested our framework on executables of benchmarks from the Polyhedral Benchmark suite [PolyBench 2010] and the NAS benchmark suite [NAS 2006]. We implemented various common dependency tests like ZIV tests (Zero induction variable), SIV test (Single induction variable), and MIV test (Multiple induction variables) as presented in [Goff et al. 1991]. We measured the number of array references where any of the dependence tests was able to eliminate dependence, or was able to provide a precise answer to the distance between dependencies. Figure 22 describes the usage and success frequency of dependence tests for each of the benchmarks. It lists the number of times a test was required to resolve the dependence between array references in each benchmark and the number of times the test was able to give a precise answer in two situations: using the memory based symbolic analysis and using only variable based symbolic analysis. Since dependence tests rely on affine expressions for loop indices, none of the dependence tests succeed when no symbolic analysis is applied. Hence, we omit the results for the case of no symbolic analysis. Figure 22 shows that the memory based symbolic analysis framework improves the precision of standard dependence tests on executables by 34% on average.

9. RELATED WORK

In this section, we discuss related work pertaining to (i) Binary Analysis (ii) Symbolic analysis (iii) Symbolic Execution and (iii) Information-flow security.

Binary analysis: There has been several binary analysis frameworks such as BitBlaze [Song et al. 2008], Jakstab [Kinder and Veith 2008], IDAPro [IDAPro disassembler], CodeSurfer/x86 [Balakrishnan and Reps 2004], BAP [Brumley et al. 2011] and others. None of these tools obtain a workable IR or perform symbolic analysis. We define a system to have workable IR if it has been demonstrated that IR can be compiled back to a working executable. Several binary rewriters such as PLTO [Schwarz et al. 2001] and UQBT [Cifuentes and Emmerik 2000] obtain a workable IR, but they have a very imprecise memory abstraction that renders them unsuitable for advanced binary analyses. Several link time optimization tools such as Diablo [De Sutter et al. 2007] obtain a workable IR but restrict their analyses to only variables. They do not make any attempts to obtain a precise abstraction for memory references. IDAPro comes the closest in trying to deal with the problem of indirect CTIs, but they do not guarantee a workable IR. Whenever `StackDiff` values cannot be determined for a particular indirect CTI, IDAPro might choose a wrong value from the pools of possible solutions resulting in an incorrect data-flow [HexBlog 2006]. Various executable frameworks ease the specification of semantics of native instructions [Thakur and Reps 2012] and address control-flow challenges [Kinder and Veith 2008], which is orthogonal to our tasks of analyzing program data flow.

Cifuentes at al [Cifuentes and Emmerik 2000] proposed an algorithm for identifying an interprocedural slice of an executable by following use-def chains, but their algorithm does not attempt to follow use-def chains in the case of memory accesses. This limits the program slice in case of memory loads.

There are various binary analysis tools [Rival 2003; Bergeron and et al. 2001; Larus and Schnarr 1995] which analyze executables in the presence of additional information such as symbol tables or debugging information. Such information is usually absent in deployed executables and our methods do not make any assumption about the presence of such extra information. In addition, none of them deal with the problem of symbolic analysis.

The work that is closely related to `ExecSVA` are frameworks proposed by Debray et al. [Debray et al. 1998], Amme et al. [Amme et al. 2000], Balakrishnan et al. [Balakrishnan and Reps 2004] and Guo et al. [Guo et al. 2005]. Debray et al. [Debray et al. 1998] and Amme et al. [Amme et al. 2000] present alias-analysis algorithms for executables. However, their biggest limitation is that they do not track memory locations and hence, lose a great deal of precision at each memory access. Balakrishnan et al. [Balakrishnan and Reps 2004] and Guo et al. [Guo et al. 2005] present memory analysis algorithms that find an over-approximation of the set of constant and memory address ranges that each abstract data object can hold. However, as presented in Section 1, symbolic analysis entails representation of values of program variables as symbolic expressions in terms of previously defined symbols. This representation enables typical symbolic analysis applications such as parallelization. The above methods. [Balakrishnan and Reps 2004; Guo et al. 2005] obtain a different representation which is not suitable for symbolic analysis applications. Further, the IR recovered by these frameworks is not workable.

Recently, there has been some work on parallelizing binary executable code. Kotha et al [Kotha et al. 2010] present a method to automatically parallelize executables using a binary rewriter. They adapt source-level affine parallelization methods for executables. Yardimci and Franz [Yardimci and Franz 2006] present non-affine automatic parallelization in a binary rewriter. Symbolic analysis methods proposed in our paper will further improve the efficiency of all these parallelization efforts by improving data dependence queries, thereby exposing more parallelism in programs. In addition, the above methods [Kotha et al. 2010; Yardimci and Franz 2006] implement custom techniques to recognize induction variables from binary executable code. Our symbolic analysis framework can obviate the need for any custom induction variable recognition method. Kotha et al [Kotha et al. 2015] present a more advanced method of improving parallelization of binary executables using cache analysis. These cache analysis techniques are orthogonal to induction variable analysis method and can be applied to any underlying binary parallelization system.

Symbolic Analysis: There has been an extensive body of work employing symbolic analysis for analyzing and optimizing programs. Various techniques broadly differ in the symbolic abstraction which is maintained as part of their analysis. Cousot and Halbwachs [Cousot and Halbwachs 1978] proposed an early method of using abstract interpretation to discover linear relationships between variables. Patterson [Patterson 1995] and Harrison [Harrison 1977] present methods for computing value ranges of program variables and employ them for improving static branch prediction [Patterson 1995]. Rugina et al [Rugina and Rinard 2000] employ symbolic constraint solvers to determine the bounds of each variable in terms of its symbolic values at the entry point of the program. Padua et al [Tu and Padua 1995] developed a system for computing symbolic values of expressions using a demand-driven backward substitution analysis on Gated-SSA form.

Symbolic analysis has been used extensively in the parallelization community to support the detection of parallelism and the optimization of programs. Haghighat et al [Haghighat and Polychronopoulos 1996] (Parafrase-2) present a symbolic analysis framework for computing a closed form expression of induction variables as well as for analyzing program properties that are essential in effective detection and exploitation of parallelization. Blume et al [Blume and Eigenmann 1994] (Polaris) present a symbolic range propagation mechanism to determine the relationship between any two arbitrary symbolic expressions by maintaining a set of symbolic range constraints for each program variable. They further employ their symbolic ranges to improve data dependence queries. The SUIF compiler [Hall et al. 2005] employs symbolic analysis to represent array indices in a symbolic form of loop index variables to apply array dependence tests. Fahringer et al [Fahringer and Scholz 1997] present a unified symbolic evaluation framework, combining both data and control flow, for determining the symbolic expressions of variables as algebraic functions over program input data.

All the above methods are source-code symbolic analysis techniques and obtain symbolic expressions for only the variables. They lose a great deal of precision when applied to binary executables directly due to the presence of memory accesses. On the other hand, we present a symbolic analysis framework for executables which tracks memory locations as well, and does not lose precision in the presence of memory accesses.

Van Put et al [Van Put et al. 2007] present a whole program linear constant analysis to analyze the stack layout of a procedure. Their analysis attempts to detect affine relations that exist between processor registers. The relations obtained through the analysis are further employed for program optimizations. However, they do not attempt to obtain any relations for memory references and each memory access operation is assumed to produce an unknown value. In contrast, we obtain a precise memory abstraction and present an analysis to track symbolic relations across memory references as well.

Symbolic execution: There has been a great deal of work on symbolic execution in the field of software testing [Cadar et al. 2008] and test case generation for security vulnerabilities [Cha et al. 2012].

Symbolic execution and symbolic analysis are similar in an aspect that both use symbolic constraints to represent values, however they are not interchangeable. Symbolic analysis, as presented in this paper, is an abstract interpretation method which determines a set of symbolic expressions for each object. On the other hand, symbolic execution generates and maintains symbolic constraints per program path and does not generalize constraints to all paths. Symbolic execution relies on constraint solvers to determine the feasibility of each path and is employed mainly for bug testing of programs.

Several tools such as Mayhem [Cha et al. 2012] have been proposed to speed up symbolic execution on executables for detecting bugs and for generating exploit inputs. They employ Value Set Analysis [Balakrishnan and Reps 2004] to reduce the load on constraint solver by resolving bounds on several variables. `ExecSVA`, as presented in our paper can potentially be employed to further reduce the load on constraint solvers employed in such symbolic execution tools. Symbolic analysis [Bodík and Anik 1998] has been employed for determining equivalence as well as for pointer analysis in source-code. `ExecSVA` can be extended to prove or disprove equivalence, thereby aiding some decisions in constraint solvers.

Information-flow security: There has been a large number of research tools for tracking information-flow at runtime. Schwartz et. al. [Schwartz et al. 2010] present an extensive survey discussing various dynamic taint mechanisms and their respective limitations. Being dynamic mechanisms, these methods only track information-flow along the execution path and are unable to provide a complete code coverage guarantee. On the other hand, `ExecSVA` and `DemandSym` are static analyses that track information-flow along the whole program.

There has been a very limited amount of work on detecting information-flow violations by statically analyzing the executable code. Major works in this approach are those suggested by Cova et. al. [Cova et al. 2006], privacy leak detection [Egele et al. 2011] and integer flow vulnerabilities [Wang et al. 2009]. A major limitation of all these methods is that they ignore memory and aliasing issues in their analysis, resulting in an imprecise vulnerability detection.

Sridharan et. al. [Sridharan et al. 2007] proposed a novel thin slicing framework for precisely capturing only the data dependencies in source-code and it has been subsequently applied [Tripp et al. 2009] for detecting information-flow vulnerabilities. As discussed in Section 2, such source-code methods cannot be applied directly in executables due to lack of source-level semantic and syntactic information.

10. CONCLUSIONS

In this paper, we have proposed techniques to obtain a workable and precise representation from executables and presented methods to adapt symbolic analysis for executables in a scalable manner. The improved memory model considerably enhances the precision of our symbolic analysis framework and novel symbolic analysis framework improves the efficacy of various analyses. In the future, we plan to extend this framework for other purposes such as binary understanding.

REFERENCES

Wolfram Amme, Peter Braun, François Thomasset, and Eberhard Zehendner. 2000. Data Dependence Analysis of Assembly Code. *Int. J. Parallel Program.* 28, 5 (Oct. 2000), 431–467.

Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013a. A compiler-level intermediate representation based binary analysis and rewriting system. In *European Conference on Computer Systems*. 295–308.

K. Anand, K Wazeer, A. Kotha, M. Smithson, and R. Barua. 2013b. A Symbolic Analysis Framework for analyzing executables. http://www.ece.umd.edu/∼barua/icsm13-extended.pdf.

Gogul Balakrishnan and Thomas Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction, 13th International Conference, CC 2004*. 5–23.

Gogul Balakrishnan and Thomas Reps. 2007. DIVINE: discovering variables in executables. In *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI*. 1–28.

D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and Giovanni Vigna. 2008. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. 387–401.

Utpal Banerjee. 1979. *Speedup of ordinary programs*. Ph.D. Dissertation. Champaign, IL, USA.

J. Bergeron and et al. 2001. Static detection of malicious code in executable programs. *Int. J. of Req. Eng* (2001).

William Blume and Rudolf Eigenmann. 1994. Symbolic Range Propagation. In *Proceedings of the 9th International Parallel Processing Symposium*. 357–363.

Rastisalv Bodík and Sadun Anik. 1998. Path-sensitive value-flow analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 237–251.

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: a binary analysis platform. In *Proceedings of the 23rd international conference on Computer aided verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 463–469.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Berkeley, CA, USA, 209–224.

Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 380–394.

Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*. 39–50.

Walter Chang, Brandon Streiff, and Calvin Lin. 2008. Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. ACM, New York, NY, USA, 39–50. DOI:http://dx.doi.org/10.1145/1455770.1455778

Cristina Cifuentes and Mike Van Emmerik. 2000. UQBT: Adaptable Binary Translation at Low Cost. *Computer* 33, 3 (2000), 60–66.

Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 84–96.

Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. 2006. Static Detection of Vulnerabilities in x86 Executables. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*. IEEE Computer Society, Washington, DC, USA, 269–278.

Crispin Cowan and et al. 2001. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *In Proceedings of the 10th USENIX Security Symposium*.

CVE. 2013. *Directory traversal vulnerability in Rack*. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0262.

CVE-MITRE. 2015. Common Vulnerabilities and Exposures. https://cve.mitre.org/.

Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: a flexible information flow architecture for software security. *SIGARCH Comput. Archit. News* 35, 2 (June 2007), 482–493.

Darpa. 2012. Announcement for Binary Executable Transforms. http://www.federalgrants.com/Binary-Executable-Transforms-BET-30063.html.

Bjorn De Sutter, Ludo Van Put, Dominique Chanet, Bruno De Bus, and Koen De Bosschere. 2007. Link-time Compaction and Optimization of ARM Executables. *ACM Trans. Embed. Comput. Syst.* 6, 1, Article 5 (Feb. 2007).

Saumya Debray, Robert Muth, and Matthew Weippert. 1998. Alias analysis of executable code. In *In Symposium on Principles of Programming Languages*. 12–24.

Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*.

Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013a. Recovering function boundaries from executables. In *Technical Report, University of Maryland, College Park*. http://www.ece.umd.edu/$\sim$barua/function-boundaries.pdf

Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013b. Scalable Variable and Data Type Detection in a Binary Rewriter. *SIGPLAN Not.* 48, 6 (June 2013), 51–60.

Thomas Fahringer and Bernhard Scholz. 1997. Symbolic Evaluation for Parallelizing Compilers. In *International Conference on Supercomputing*. 261–268.

FireEye. 2015. Recent Zero-Day Exploits. https://www.fireeye.com/current-threats/recent-zero-day-attacks.html.

Gina Goff, Ken Kennedy, and Chau-Wen Tseng. 1991. Practical dependence testing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation (PLDI '91)*. ACM, New York, NY, USA, 15–29.

Bolei Guo, Matthew J. Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I. August. 2005. Practical and Accurate Low-Level Pointer Analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*. 291–302.

Samuel Z. Guyer and Calvin Lin. 2003. Client-driven pointer analysis. In *Proceedings of the 10th International Conference on Static Analysis*. 214–236.

Mohammad R. Haghighat and Constantine D. Polychronopoulos. 1996. Symbolic analysis for parallelizing compilers. *ACM Trans. Program. Lang. Syst.* 18, 4 (July 1996), 477–518.

Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. 2005. Interprocedural parallelization analysis in SUIF. *ACM Trans. Program. Lang. Syst.* 27, 4 (July 2005), 662–731.

W. H. Harrison. 1977. Compiler Analysis of the Value Ranges for Variables. *IEEE Trans. Softw. Eng.* 3, 3 (May 1977), 243–250.

Nevin Heintze and Olivier Tardieu. 2001. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*.

HexBlog. 2006. Simplex method in IDA Pro. http://www.hexblog.com/?p=42.

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

Peter Kankowski. 2006. x86 Machine Code Statistics. http://www.strchr.com/x86_machine_code_statistics.

Johannes Kinder and Helmut Veith. 2008. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification*. 423–427.

Aparna Kotha, Kapil Anand, Timothy Creech, Khaled Elwazeer, Matthew Smithson, Greeshma Yellareddy, and Rajeev Barua. 2015. Affine Parallelization Using Dependence and Cache Analysis in a Binary Rewriter. *IEEE Trans. Parallel Distrib. Syst.* 26, 8 (2015), 2154–2163.

Aparna Kotha, Kapil Anand, Matthew Smithson, Greeshma Yellareddy, and Rajeev Barua. 2010. Automatic Parallelization in a Binary Rewriter. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. 547–557.

James R. Larus and Eric Schnarr. 1995. EEL: machine-independent executable editing. *SIGPLAN Not.* 30, 6 (June 1995), 291–300.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *In IEEE/ACM International Symposium on Code Generation and Optimization*. 75–87.

Cullen Linn and Saumya Debray. 2003. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*. ACM, New York, NY, USA, 290–299.

NAS. 2006. NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html/.

Oink. 2015. *Oink Tool*. http://daniel-wilkerson.appspot.com/oink/index.html//.

Jason R. C. Patterson. 1995. Accurate static branch prediction by value range propagation. *SIGPLAN Not.* 30, 6 (June 1995), 67–78.

PolyBench. 2010. Polybench: The Polyhedral Benchmark Suite. http://www-roc.inria.fr/~pouchet/software/polybench/.

Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings (Lecture Notes in Computer Science)*, Bernhard Steffen and Giorgio Levi (Eds.), Vol. 2937. Springer, 252–266.

Xavier Rival. 2003. Abstract Interpretation-Based Certification of Assembly Code. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2003)*. Springer-Verlag, London, UK, UK, 41–55.

Radu Rugina and Martin Rinard. 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *SIGPLAN Not.* 35, 5 (May 2000), 182–195.

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. 317–331.

Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. 2001. PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture. In *In Proc. 2001 Workshop on Binary Translation*.

Setuid. Setuid: Linux Man Page. http://linux.die.net/man/2/setuid.

Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. 2001. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium,2001*. 16–16.

Matthew Smithson, Khaled Elwazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. 2013. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*. 52–61.

Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS '08)*. Springer-Verlag, Berlin, Heidelberg, 1–25.

SourceForge. 2013. Calling Conventions. http://asm.sourceforge.net/howto/conventions.html.

Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07)*. ACM, New York, NY, USA, 112–122.

TCMalloc. TCMalloc: Thread Caching Malloc. http://www.goog-perftools.sourceforge.net/doc/tcmalloc.html.

Aditya Thakur, Junghee Lim, Akash Lal, Amanda Burton, Evan Driscoll, Matt Elder, Tycho Andersen, and Thomas Reps. 2010. Directed Proof Generation for Machine Code. In *Proceedings of the 22Nd International Conference on Computer Aided Verification (CAV'10)*. Springer-Verlag, Berlin, Heidelberg, 288–305. DOI:http://dx.doi.org/10.1007/978-3-642-14295-6_27

Aditya Thakur and Thomas Reps. 2012. A method for symbolic computation of abstract operations. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*. 174–192.

Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *Proceedings of the ACM SIGPLAN 2009 conference on Programming Language design and Implementation*. 87–97.

Peng Tu and David Padua. 1995. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th international conference on Supercomputing (ICS '95)*. ACM, New York, NY, USA, 414–423.

Ludo Van Put, Dominique Chanet, and Koen De Bosschere. 2007. Whole-program Linear-constant Analysis with Applications to Link-time Optimization. In *Proceedingsof the 10th International Workshop on Software and Compilers for Embedded Systems (SCOPES '07)*. ACM, New York, NY, USA, 61–70.

Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2008. STILL: Exploit Code Detection via Static Taint and Initialization Analyses. In *Computer Security Applications Conference, Annual*. 289 –298. DOI:http://dx.doi.org/10.1109/ACSAC.2008.37

Michael Joseph Wolfe. 1990. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA.

Efe Yardimci and Michael Franz. 2006. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *Proceedings of the 3rd conference on Computing frontiers (CF '06)*. ACM, New York, NY, USA, 127–138.

Jingbo Zhang, Rongcai Zhao, and Jianmin Pang. 2007. Parameter and Return-value Analysis of Binary Executables. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*. 501–508.