2013 IEEE International Conference on Software Maintenance

An Accurate Stack Memory Abstraction and Symbolic Analysis Framework for Executables

Kapil Anand, Khaled Elwazeer, Aparna Kotha, Matthew Smithson, Rajeev Barua
University of Maryland
College Park
{*kapil,wazeer,akotha,msmithso,barua*}*@umd.edu*

Angelos Keromytis
Columbia University
New York
angelos@cs.columbia.edu

***Abstract*—This paper makes two contributions regarding reverse engineering of executables. First, techniques are presented for recovering a precise and correct stack memory model in executables in presence of executable-specific artifacts 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 analysis as source-level tools. Frameworks hitherto fail to simultaneously maintain the properties of correct representation and precise memory model and ignore memory-allocated variables while 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 by applying it to various traditional analyses, including identifying information flow vulnerabilities in five real-world programs.**

I. 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 several popular applications. It is a well-known fact that most of the applications used on a daily basis are IP-protected software that are available only in the form of executables. Robust reverse-engineering tools are required to completely analyze the impact of latest cyberattacks on such applications, to define efficient counter strategies and to certify their robustness against such attacks.

Reverse engineering tools are also essential for continuous software maintenance. Various organizations such as US Department of Defense [1] have critical applications that have been developed for older systems and need to be ported to future secure versions in light of exposed vulnerabilities. In many cases, the application source code is no longer accessible requiring these applications to continue to run on outdated configurations. This engenders a need for advanced tools which enable identification and extraction of functional components for reuse in new applications.

The applicability of a reverse-engineering framework in the above scenarios of vulnerability detection, software certification and software maintenance entails three desired features: 1) The recovered intermediate representation (IR) should be *functional* such that it can be employed to recover a functional source code or recompiled to obtain a working rewritten binary. A non-functional code also fails to capture the complete application behavior, resulting in inaccurate analysis results. 2) Since executables mainly contain memory locations instead of explicit program variables, the IR should have a *precise memory abstraction* to effectively reason about memory operations in presence of executable-specific features such as indirect control transfer instructions (CTI) and lack of procedure prototypes. The paucity of registers in x86 ISA further underscores this requirement by allocating most of the variables to 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*. Unfortunately, obtaining all three features in a framework is very challenging when dealing with stripped binaries which do not contain symbolic or debugging information.

Executable specific artifacts such as indirect CTIs complicate the task of recovering a *precise memory abstraction* while maintaining the *functionality* in IR. A 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 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, including CTIs which can possibly modify the stack pointer due to several reasons such as cleanup of arguments.

However, the modification in the value of stack pointer cannot be easily determined in all scenarios. For example, in case of an indirect CTI, the stack modification is deterministic only if all its statically determined possible targets modify the stack pointer by the same value. However, such targets might modify the stack pointer by different values, or a call to an external function with an unknown prototype might have a statically indeterminable impact on the value of stack pointer. Existing frameworks [2], [3] require that the return from a CTI should always modify the stack pointer by a deterministic constant value.

We present techniques for recovering a precise memory

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.20

model and functional IR in such scenarios. Our mechanism formulates a set of constraints using control flow constructs in the caller procedure to compute the value of stack modification at a call-site. The constraints are solvable in most scenarios. When the constraints cannot be solved, it embeds run-time checks to maintain the functionality of IR.

This enhanced memory model improves the precision of several analysis techniques for executables. In our second contribution, we employ this memory model and present a novel symbolic analysis for executables, *Symbolic Value Analysis*, which enables *analysis similar to source code*. Symbolic analysis [4], [5] is employed for a variety of applications such as alias analysis and security analysis. In source code, only pointer and array accesses are considered memory accesses, hence such symbolic analysis methods [4], [5] only focus on instructions involving program variables. Due to a large percentage of memory operations in executables, a symbolic analysis framework for executables must employ a precise memory model.

However, existing frameworks either ignore memory models while recovering a symbolic abstraction or do not recover a symbolic abstraction. Several executable techniques [6], [7], [8] restrict their analysis to only the registers and handle memory locations in a very conservative manner. Consequently, these methods lose a great deal of precision at each memory access. On the other hand, several popular analysis frameworks for executables, notably Value Set Analysis (VSA) [3] and related methods [9], analyze memory accesses but represent values of program variables as a set of integral values and memory locations, which do not represent the symbolic relations between these variables.

The primary contributions of our work are the following:

- → **Precise and correct stack memory abstraction**: We present a hybrid static-dynamic mechanism to determine the impact of executable specific artifacts such as indirect CTIs on the value of stack pointer, resulting in a functional representation and a precise memory model. This mechanism can be employed to improve the precision of any existing memory analysis framework such as VSA [3] and others [9].
- → **Novel Symbolic Analysis**: Based on the improved memory model, we formulate a novel Symbolic Value Analysis which computes symbolic abstraction for variables as well as for memory locations.
- → **Applications**: We extend our analysis for several applications such as security analysis, redundancy removal and demonstrate that client applications become less effective when memory tracking is not enabled.

We evaluated our techniques with SPEC2006 benchmark suite as well as several real world programs such as Apache server. Our techniques improve the precision of memory models by 25% in programs containing significant number of indirect CTIs. This improved memory model enhances the precision of Symblic Value Analysis by 20% on average.

```
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)
    ......
```

Figure 1: *An example demonstrating the imprecision in the presence of indirect calls, second operand in the instruction is the destination*

Our techniques are scalable and analyze large programs such as *gcc* in less than 5 minutes.

II. Motivation

In this section, we demonstrate the limitation of existing frameworks in obtaining a functional IR with a precise memory model and the relative importance of considering the underlying memory model for symbolic abstraction.

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 IR, the physical stack must be deconstructed to individual abstract frames per procedure. Since, each such frame comprises 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 [3] have approached this problem in two steps. First, all the instructions in a procedure which 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 abstract 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 determination of the value of the stack pointer at each program point in a procedure relative to its value at the entry point.

As highlighted in Section I, 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 [3]. We characterize the impact of a CTI `I` on the value of stack pointer using the following definition:

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

The term `StackDiff` can be applied to either the CTI or a corresponding called procedure, and represents the stack modification amount in either case. `StackDiff` of a CTI can be 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. Several approaches have been suggested to calculate the value of `StackDiff` by symbolically evaluating all the stack modification instructions in a procedure [3]. As per these methods, `StackDiff` at an 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, 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 scenarios, existing frameworks either result in an imprecise memory abstraction or fail to maintain the correctness. As per CodeSurfer/X86, "if it cannot determine that the change is a constant, it issues an error report" (Section 4.2) [3]. Hence, the corresponding frame cannot be represented through `a-locs`, resulting in an imprecise memory model. IDAPro applies a constraint-based mechanism to compute the values of `StackDiff` independent of the called procedures. However, when the underlying method fails to determine a unique solution, it compromises the correctness by accepting one feasible solution (which could be wrong) out of an infinite number of possible outcomes [10].

Fig 1 illustrates an example of such a scenario. In Fig 1, a local region of size 24 is allocated in a procedure, 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 `StackDiff` value is calculated wrongly, it results in an incorrect data-flow at Line 4.

Our hybrid mechanism maintains the precision as well as functionality. Our static mechanism enables abstraction through a set of `a-locs` and dynamic mechanism guarantees the correctness when `StackDiff` cannot be computed.

Symbolic abstraction: Since executables extensively employ memory locations, not analyzing them for symbolic analysis in executables results in imprecise symbolic relations. Fig 2(a) shows a source code example and the relations between various computations determined through symbolic analysis. Fig 2(b) shows a sample code which 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. *Since existing symbolic analyses for source code [4] as well as for executables [6] do not propagate symbolic expressions across memory locations, a new symbol is defined at every memory reference instruction.*

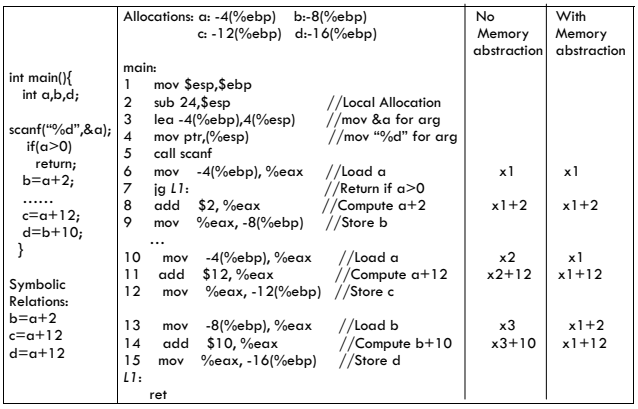

Figure 2: *(a) A sample C code (b) Corresponding assembly code, the second operand in the instruction is the destination (c) Symbolic relations on the assembly code with no memory abstraction (d)Symbolic relations on the assembly code with memory abstraction*

As evident from Fig 2(c), the resulting symbolic relations are conservative and yield imprecise program information.

We observe that representing a symbolic abstraction for memory locations can eliminate this limitation. Fig 2(d) shows the symbolic relations when a symbolic abstraction is maintained for memory locations as well. Suppose, the variable `a (-4(%ebp))` has value `x1` in the enviroment 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`. Similarily, 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 through the source code.

III. Overview

Fig 3 presents an overview of our binary analysis framework. Our framework is built over existing SecondWrite framework as presented in [11]. SecondWrite translates the input x86 binary code to a functional program represented in the intermediate representation (IR) of the LLVM Compiler [12]. SecondWrite implements various mechanisms to obtain an IR which contains features like 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 [2], [3] can use our analysis*. LLVM IR obtained above is passed through our analysis system.

A key challenge in binary analysis is discovering which portions of the code section in an input executable are definitely code. SecondWrite implements *speculative disassembly* and *binary characterization*, proposed by Smithson and Barua [13], to efficiently address this problem. The indirect CTIs are handled by translating the original target to the corresponding location in IR through a *call translator*

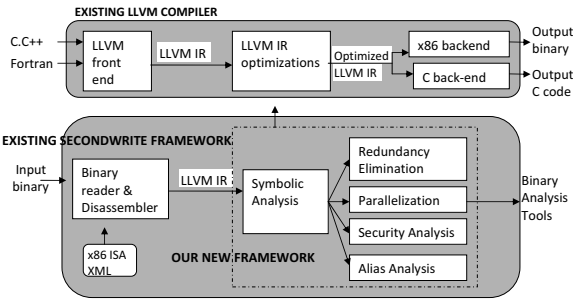

Figure 3: *Organization of the system*

procedure [13]. Each recognized procedure (through speculative disassembly) is initially considered a possible target of the translator, which is pruned further using alias analysis.

Memory assumptions: Similar to most executable analysis frameworks [3], [14], [15], our techniques assume that executables follow the *standard compilation model* where each procedure may allocate an optional stack frame in one direction only and each variable resides at a fixed offset in its corresponding region. Like most static binary tools, we do not handle self modifying or obfuscated code.

IV. Recovering Precise Memory Model

In this section, we discuss our hybrid static-dynamic solution to obtain a functional representation with a precise 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 functionality even when `StackDiff` at some CTIs cannot be solved. Our analysis employs the prototypes of well-known library functions, similar to the IDAPro's FLIRT database [2], for determining their `StackDiff` value. We assume that existing methods [3] are able to determine the value of `StackDiff` for each procedure, which holds true under the assumptions of *standard compilation model*.

A. 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`.

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 extremely challenging problem.

We define symbolic values X_I and S_I for representing `StackDiff` and local stack height at a CTI `I`. Every stack

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 address T
`StackDiff(f)`: StackDiff of procedure `f`
`Y_SET(F)` = $\cup_{f \in F}$ `StackDiff(f)`
`BeginP` = Entry point of procedure `P`; $Pred_{BB}$ = Predecessors of basic block `BB`;
`BeginBB, EndBB` = Entry point,terminator of basic block `BB`
S_I = Stack height after instruction `I`;
S_{BB} = Stack height at beginning of basic block `BB`;
`PrevI` = the previous instruction to `I` (`I` $\neq$ `BeginBB`)
$S_{I'}$ = if (`I` $\neq$ `BeginBB`) then S_{PrevI} else S_{BB}
`R` : A register, `Size(R)`: Size of register `R`, `N`: A constant

Initial Conditions : $S_{BeginP} = 0$

Data flow rules :

For every instruction `I`:

`I` = push R $\Rightarrow$ $S_I = S_{I'}$ + `size(R)`
`I` = pop R $\Rightarrow$ $S_I = S_{I'}$ - `size(R)`
`I` = add esp, N $\Rightarrow$ $S_I = S_{I'}$ - N
`I` = sub esp, N $\Rightarrow$ $S_I = S_{I'}$ + N
`I` = jmp L $\Rightarrow$ $S_{BeginL} = S_{I'}$
`I` = call `Y` $\Rightarrow$
  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) $\Rightarrow$ $S_I = S_{I'}$

Boundary Conditions :

1. $\forall$ `BB`: $\forall$ `Pred` $\in$ $Pred_{BB}$, $S_{BeginBB} = S_{EndPred}$
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

Figure 4: *Data flow rules used to determine stack modifications in a procedure P*

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 the value of X_Is.

Fig 4 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 4 incorporate the deterministic modification at each CTI. An indeterministic modification is modeled symbolically as X_I. The dataflow rules in Fig 4 obtain an expression for S_I considering each such stack modification instruction.

In order to solve the above symbolic equations, Fig 4 generates two constraints based on the control flow in procedure `P`. These conditions hold true for every executable following the standard compilation model [3]:

- $\rightarrow$ $\forall$`Pred` $\in$ $Pred_{BB}$, $S_{BeginBB} = S_{EndPred}$: 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.

→ $S_{I'} = 0 \ \forall$ ret $\in$ 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. Since the return address is pushed by the caller procedure and a compiled program usually does not access the return address directly, 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 some incoming arguments, so StackDiff could be positive or zero.

Fig 4 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 the correctness.

→ I = esp ← ... : 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.[1]

→ 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 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 [3].

B. 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 handle all the three cases of *Unknown CTIs* presented in Section IV-A.

[1]Code produced by popular compilers contains x86 idioms like leave instruction which implicitly assign a previously stored value to esp. Such idioms are currently handled explicitly in our framework.

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

Figure 5: *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*

Case 1: Since this case represents control transfer to an external procedure, the body of the called procedure cannot be modified. Such scenarios are handled by calling the external procedure using a trampoline. The trampoline dynamically computes the shift in stack pointer value before and after the call using inline assembly instructions.

Case 2 and **Case 3**: Recall from Section III, 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.

V. Symbolic Value Analysis

Our technique, Symbolic Value Analysis, is a flow-sensitive, context insensitive analysis which computes a conservative over-approximation of a set of symbolic values that each data object (variables and a-locs) can hold at each program point. Symbolic Value Analysis is based on memory model obtained in Section IV.

Sec V-A first presents the abstraction for representing the symbolic values in our analysis and subsequent sections discuss the intraprocedural and interprocedural versions of the analysis.

A. Symbolic Abstraction

Fig 5 presents the grammar for representing the symbolic expressions in our abstraction. As evident from Fig 5, symbolic expressions are numeric algebraic polynomials containing sums of product terms of variables.

Symbolic Value Set: A symbolic value set is a finite set of symbolic expressions defined by the Grammar in Fig 5. It constitutes a conservative over-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 ($\otimes$). The abstraction also supports a Widen (∇) operator. This operator implements the inherent widening operation in our environment. If the required cardinality increases beyond a limit, we invalidate the current symbolic value set. This operation prevents the exponential blowup of symbolic expressions.

$$\nabla \texttt{SymValSet}_1 = \{\texttt{if}\,|\texttt{SymValSet}_1| > \texttt{LIMIT}, \texttt{then}\ \top \\ \texttt{else SymValSet}_1\} \quad (1)$$

B. Intraprocedural Analysis

Our analysis defines three kind of memory regions, associated with procedures (Stack), global data (Global) and heaps (HeapRgn). The method presented in Section IV enables us to precisely abstract the above memory regions through a set of a-locs in the presence of indirect CTIs.

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.

SM_e: Map between a-locs and their corresponding symbolic value sets before a program point e

Executables regularly employ the *indirect-addressing* mode for accessing memory locations. After obtaining a-locs using the framework in Section IV, VSA [3] 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 we present our algorithm 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.

R_i: IR (SSA) variables; e : A program point; r: Data object (Var or a-loc)

SM'_e: Map between a-locs and their symbolic value sets after program point e

SR(r): Mapping of Var r in map SR; SM_e(r): Mapping of a-loc r in map SM_e

Mem_e(r):Set of memory addresses that r can hold at point e (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 $\mathrm{Mem}_e(r)$ contains a list of memory addresses that the data object r can hold at current program point e. 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 Mem_e(r), represents the fully accessed a-locs. The partially accessed a-locs consists (i) a-locs whose starting addresses are in Mem_e(r) but are not of size s and

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

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

(ii) a-locs whose addresses are in Mem_e(r) but whose starting addresses and size do not meet the condition to be fully accessed a-locs. Using the notation from [3], this operation is mathematically represented as:

{F,P} = *(Mem_e(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 em-

ploying 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 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 updated 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.

C. Interprocedural propagation

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

Several binary analysis frameworks [16], [3], including SecondWrite [11], 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

$$SR = \{SR - SR(f_i)\} \cup \{(f_i, \nabla(\bigcup_{\forall c \in CallSites(P)} SR(a_{ci})))\} \quad (2)$$

The return variables are also handled in a similar manner. In order to propagate the symbolic values of memory locations, 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.

$$SM_{P_{entry}} = \bigcup_{\forall c \in CallSites(P)} SM_c \quad (3)$$

Similarly, 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*.

The 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 since 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.

VI. Results

Our techniques are implemented as part of the SecondWrite framework presented in Section III. The evaluation is performed on several benchmarks from the SPEC2006 and OMP2001 suites and some real world programs, as listed in Table II. 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.

A. Functional Representation and Precise Memory Model

Fig 6 and Fig 7 present the statistics regarding our hybrid mechanism for obtaining precise memory model and functional IR. We only present statistics for benchmarks containing non-negligible *Unknown CTIs* (negligible defined as ≤ 10 or number of procedures containing *Unknown CTI* $\leq 1\%$). Of 33 programs in Table II, 11 had non-negligible unknown CTIs. Fig 6 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 functionality. Case 1 (Section IV-A) does not arise since we employ the prototypes for standard library procedures. We never hit the invalidation conditions stipulated in Fig 4, justifying the assumptions behind our formulation.

Fig 7 illustrates the additional a-locs derived as a result of successful constraint solutions, normalized with respect to original a-locs of type `Stack` (Section V-B). As evident, we were able to obtain 10% more `a-locs` in C benchmarks and 30% more `a-locs` in C++ benchmarks on average.

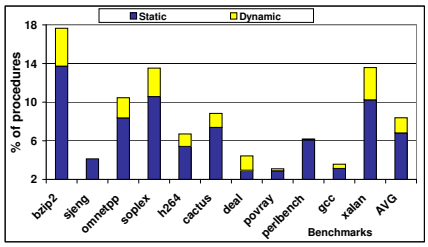

Figure 6: *Percentage of procedures with unknown CTIs. The static represents cases when constraint solvers succed*

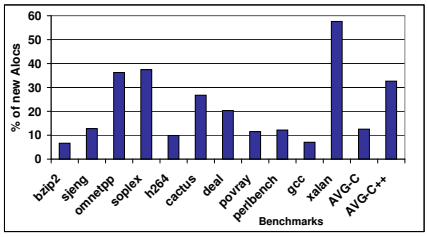

Figure 7: *Additional alocs added as a result of constraint solvers, normalized to original number of alocs*

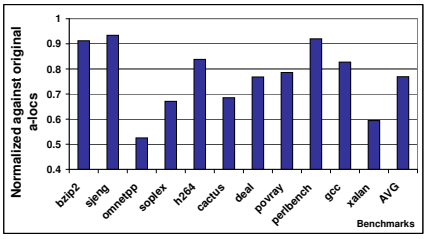

Figure 8: *Variables requiring a new symbolic alphabet in presence of additional a-locs*

Application	Source	Lang	LOC	# Proc	Time(s)	Mem (MB)
bwaves	Spec2006	F	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	F	2468	43	1.37	5.68
libquantum	Spec2006	C	2743	73	1.30	6.30
leslie3d	Spec2006	F	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	F	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/F	65182	674	47.82	252.33
dealII	Spec2006	C++	96382	15619	114.30	240.18
calculix	Spec2006	C/F	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	280.37	490.68
xalan	Spec2006	C++	267318	30,062	264.97	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

Table II: *Applications Table*

This enhanced `a-locs` abstraction is employed in our symbolic value analysis framework.

B. Symbolic Value Analysis

Table II shows the analysis time and storage requirements of our Symbolic Value Analysis 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 number of procedures in the benchmark. The analysis time is typically low, within 1 minute, for most of the benchmarks except for some intensive benchmarks such as gcc and dealII.

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. We observe that 35% of symbolic expressions contain alphabets propagated through memory locations. An extended version of the paper [17] presents more detailed results. 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. On average, 64% of objects can be abstracted with a single symbolic expression in our symbolic domain, 16% of objects need multiple expressions and 20% of objects cannot be represented with finite symbolic abstraction ($\top$) [17]. Maintaining a symbolic value set instead of a single symbolic expression allows us to maintain this extra precision for 16% of data objects.

Fig 8 captures the enhancement in the precision of Symbolic Value Analysis with the presence of additional `a-locs` derived by the constraint mechanism. According to Table I, a load instruction accessing an unknown memory location is represented by a new symbolic alphabet. Fig 8 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.

C. Applications

As mentioned before, symbolic analysis is employed in multiple source-level analysis. Here, we demonstrate that our Symbolic Value Analysis enables us to extend several source-level analyses to executables. An extended version [17] explores more applications such as parallelization and alias analysis.

Value numbering: It has been shown that even highly optimized executables contain large amount of redundant instructions. For example, Fernandez et al. [18] observed that around 30% of memory references in an optimized program are redundant. Redundancy elimination simplifies the intermediate representation, thereby aiding other optimizations

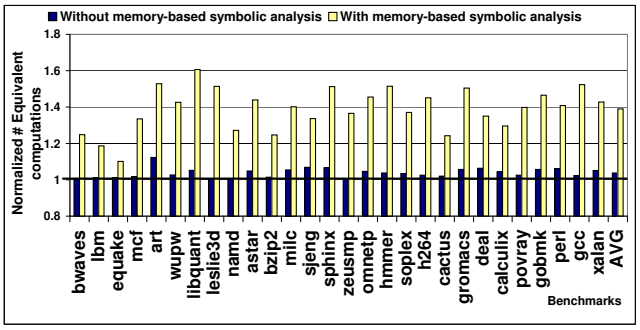

Figure 9: *Normalized improvement in detection of equivalent computations (No Symbolic analysis = 1.0)*

and speeding up subsequent binary analysis. For example, time taken by bug testing tools for solving a query can be cut in half by simplifying the query first [19].

As explained in Section II, memory based symbolic analysis frameworks obtain more complete symbolic relations between computations in an executable, exposing more equivalences than defined by traditional Value numbering.

We define an abstract interpretation based algorithm on the lattice of symbolic expressions. Fig 9 compares the number of equivalent computations determined in three cases: one when no symbolic analysis is performed, second when symbolic analysis is employed only for variables (obtained by neglecting the transfer functions for memory load and memory store in Table I) and third, when memory based symbolic analysis is employed to determine equivalence. Hence, the second case is similar to existing source-level methods of symbolic analysis since it tracks only variables. As evident, numbering employing memory-based symbolic analysis is able to expose around 40% more equivalent computations in executables than base value numbering (when no symbolic analysis is applied). This figure also shows that symbolic analysis based only on variables is not sufficient in exposing more equivalences in executables and exposes only 3% more equivalences than discoverable when no symbolic analysis is applied. This underscores the importance of maintaining symbolic abstraction for memory locations in improving the efficacy of the applications.

Security analysis : Information flow violations represent one of the most serious security challenges in modern software systems. Source-level information-flow frameworks [20] employ scalable thin slicing to accurately reason about information propagation in the presence of pointers. However, the lack of a scalable framework to accurately track memory locations in an executable forces the existing executable analyses to ignore memory references [21], resulting in an imprecise detection of violations. Consequently, most of the executable level frameworks resort to dynamic information-flow analysis for detecting the violations. Our framework statically detects information flow violations in executables with a high degree of precision and a small number of false alarms.

Program	Exploit Ref	Type	False Alarms	False Alarms (Source Tools)
muh	CAN-2000-0857	Format String	0	0
pfingerd	NISR161220028	Format String	2	0
wu-ftpd	CVE-2000-0573	Format String	0	6
gzip	CVE-2005-1228	Dir Traversal	1	0
tar	CVE-2001-1267	Dir Traversal	0	0

Figure 10: *Security Analysis*

Here we evaluate our framework for detecting two important security flaws namely *format string vulnerability* and *directory traversal vulnerability* [22]. Format string flaws arise due to an unsafe implementation of variable-argument functions like *printf* in C library. In such functions, a *format string* argument specifies the number and type of other arguments. However, there is no runtime routine to verify that the function was actually called with the arguments specified by the *format string*. Similarly, a directory traversal vulnerability arises when a filename supplied by an user is employed in a file-access procedure without sufficient validation. Intuitively, information flow violations can be detected by checking the presence of insecure values in the symbolic expressions corresponding to a sensitive variable.

In order to detect these two vulnerabilities, we define the user input functions as *corrupting* functions and variable argument functions and file open functions as *sensitive* functions respectively. The presence of a symbolic alphabet, defined at a callsite of any *corrupting* function, in the symbolic value set corresponding to underlying argument of any *sensitive* function signifies a vulnerability.

Fig 10 shows the evaluation of our method on five programs with known vulnerabilities. As evident, we are able to detect these vulnerabilities, reporting fewer false alarms than source-level tools [22]. Our method fails to detect any of the vulnerabilities if the symbolic propagation across memory locations is disabled.

VII. Related Work

Binary analysis: There has been several binary analysis frameworks such as BitBlaze [19], Jakstab [23], IDAPro [2], CodeSurfer/x86 [3] and others. None of these tools obtain a functional IR or perform customized symbolic analysis. Several binary rewriters such as PLTO [15], UQBT [8] obtain a functional IR, but it has very imprecise memory abstraction, which is not suitable for advanced binary analyses. As described in Section II, IDAPro comes the closest in trying to deal with the problem of indirect CTIs, but they do not guarantee a functional IR.

The work that is closely related to symbolic value analysis are frameworks proposed by Debray et al. [6], Amme et al. [7], Balakrishnan et al. [3] and Guo et al. [9]. Debray et al. [6] and Amme et al. [7] 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. [3] and Guo et al. [9] 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 II, such an abstraction is not suitable for symbolic analysis applications. Further, the IR recovered by these frameworks is not functional.

Symbolic Analysis: There has been an extensive body of work employing symbolic analysis for analyzing and optimizing programs. Cousot [24] proposed an early method for discovering the linear relationships between variables. Rugina et al [25] employ symbolic constraint solvers to determine symbolic bounds of each variable. Symbolic analysis has been used extensively to support the detection of parallelism [4]. However, all these above methods obtain symbolic expressions for only the variables and not memory locations, hence they lose a great deal of precision when applied to executables.

Value numbering: Several source code algorithms to discover equivalences are based on an algorithm by Kildall [26]. Later, Bodik et al [5] and others proposed more precise algorithms using backward symbolic propagation and path sensitive analysis. However, all these algorithms are based on variables alone and none of these variables propagate value numbers across memory locations.

VIII. Conclusions

In this paper, we have proposed techniques to obtain a functional and precise representation from executables and presented methods to adapt symbolic analysis for executables. 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

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

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

[3] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *In CC'04*, pp. 5–23.

[4] M. R. Haghighat and C. D. Polychronopoulos, "Symbolic analysis for parallelizing compilers," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, pp. 477–518, Jul. 1996.

[5] R. Bodík and S. Anik, "Path-sensitive value-flow analysis," in *POPL '98*, pp. 237–251.

[6] S. Debray, R. Muth, and M. Weippert, "Alias analysis of executable code," in *POPL '98*, pp. 12–24.

[7] W. Amme, P. Braun, F. Thomasset, and E. Zehendner, "Data dependence analysis of assembly code," *Int. J. Parallel Program.*, vol. 28, no. 5, pp. 431–467, Oct. 2000.

[8] C. Cifuentes and M. V. Emmerik, "Uqbt: Adaptable binary translation at low cost," *Computer*, vol. 33, no. 3, pp. 60–66, 2000.

[9] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August, "Practical and accurate low-level pointer analysis," in *CGO '05*, pp. 291–302.

[10] Simplex method in IDA Pro, http://www.hexblog.com/?p=42.

[11] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *EuroSys '13*, pp. 295–308.

[12] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–87.

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

[14] G. Balakrishnan and T. Reps, "Divine: discovering variables in executables," in *VMCAI'07*, pp. 1–28.

[15] B. Schwarz, S. Debray, G. Andrews, and M. Legendre, "PLTO: A link-time optimizer for the intel ia-32 architecture," in *In Proc. 2001 Workshop on Binary Translation*, 2001.

[16] J. Zhang, R. Zhao, and J. Pang, "Parameter and return-value analysis of binary executables," in *COMPSAC '07*, pp. 501–508.

[17] A Symbolic Analysis Framework for analyzing executables, http://www.ece.umd.edu/∼barua/icsm13-extended.pdf.

[18] M. Fernández, R. Espasa, and S. K. Debray, "Load redundancy elimination on executable code," in *Euro-Par '01*, pp. 221–229.

[19] D. Song and et al., "Bitblaze: A new approach to computer security via binary analysis," in *ICISS '08*, pp. 1–25.

[20] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," in *PLDI '09*, pp. 87–97.

[21] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, "Static detection of vulnerabilities in x86 executables," in *ACSAC '06*, pp. 269–278.

[22] S. Z. Guyer and C. Lin, "Client-driven pointer analysis," in *SAS'03*, pp. 214–236.

[23] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *CAV '08*, pp. 423–427.

[24] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *POPL'78*, pp. 84–96.

[25] R. Rugina and M. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," *SIGPLAN Not.*, vol. 35, no. 5, pp. 182–195, May 2000.

[26] G. A. Kildall, "A unified approach to global program optimization," in *POPL '73*, pp. 194–206.