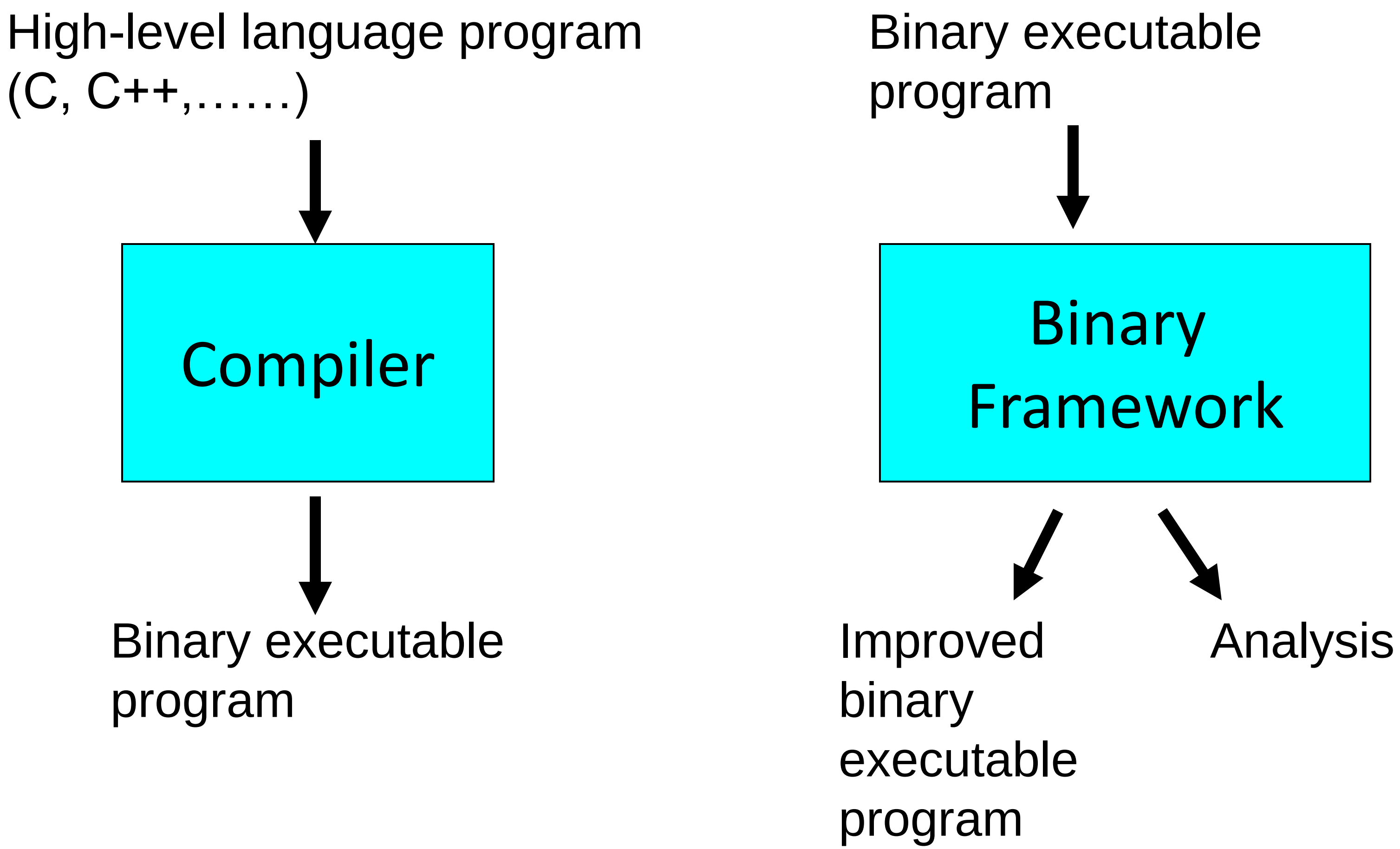
Stack Memory Abstraction and Symbolic Analysis Framework for Executables

Kapil Anand
Khaled Elwazeer
Aparna Kotha
Matthew Smithson
Rajeev Barua
Angelos Keromytis

Binary Framework



Applications

- Analyses of vulnerable code
- Malware Analysis
- End-user security enforcements
  - Custom security policies
- Platform specific optimizations
  - Memory hierarchy
  - Multimedia instructions

Existing Binary Tools

- Significantly lag behind compilers in capability
  - No complex analyses

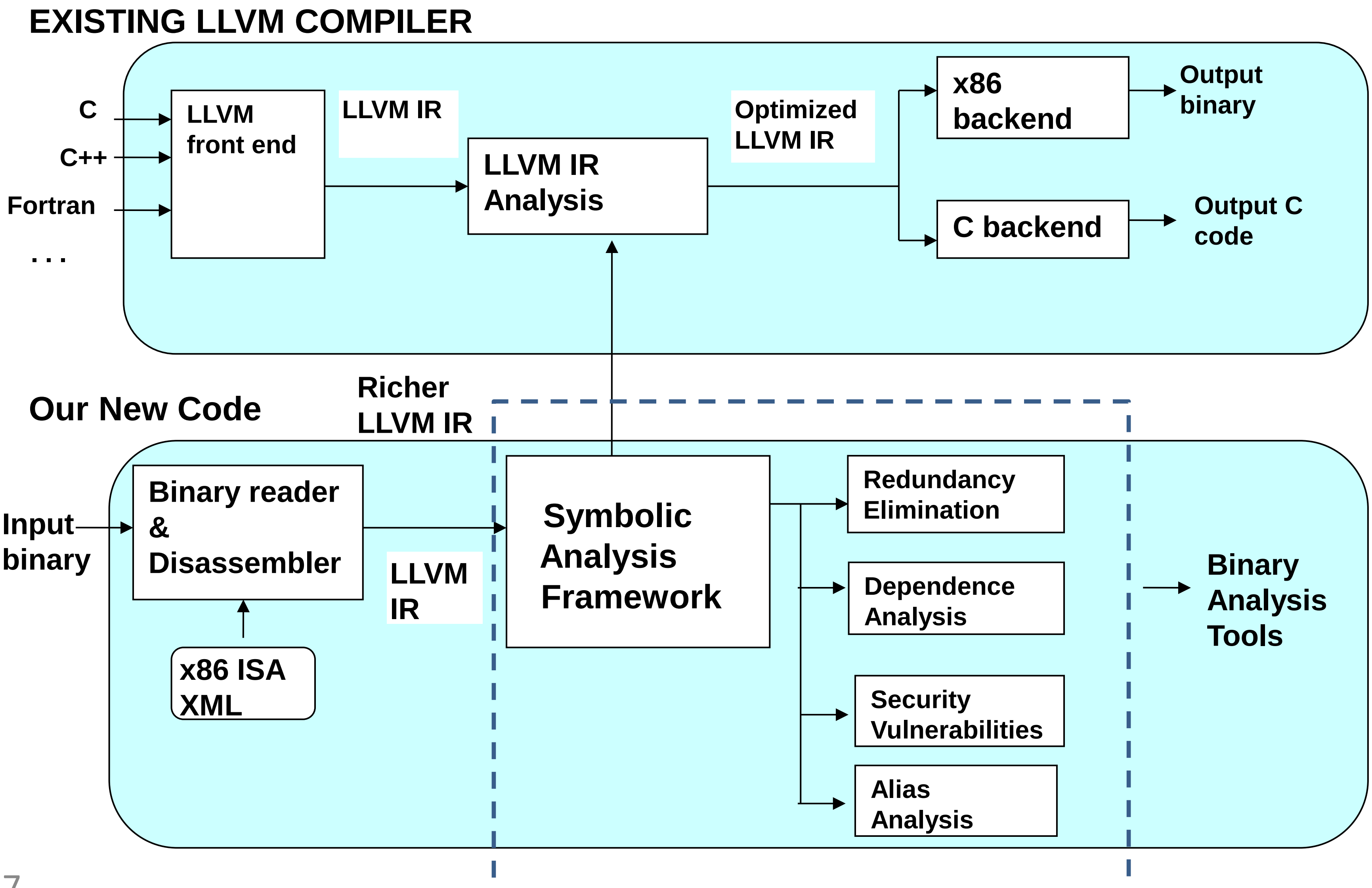
Goals

- Static analysis
  - Limitations of dynamic frameworks
    - No time to do complex analyses
    - Limited code coverage
- Functionality
  - Ability to rewrite code correctly and obtain functional representations
- High-level Intermediate Representation (IR)
  - No extra constraints in comparison to source-code
- Practicality
  - No use of meta-data information
- Scalability
  - Applicable to real-world programs

Assumptions

- Compilation memory model (Reps,2004)
- No self modifying/dynamically generated code
- No obfuscated code
- Disassembly assumptions
  - No calculated addresses (WCRE, 2013)

SecondWrite



Challenges

- Underlying analyses are not *capable* to reach source-level analyses.
- Presence of executable artifacts limits *representation* and *functionality.*

Analysis in Executables

- Source-level frameworks
  - Capability: Precision of underlying analyses
  - Only analyze instructions involving program variables.

- Existing binary frameworks
  - Ignore memory models

- Example: Symbolic Analysis
  - Represents the values of program variables as symbolic expressions
  - Applications: Value numbering, dependence

Limitations of source level analysis: Symbolic Analysis

	Allocations: a: -4(%ebp) b:-8(%ebp) d:-16(%ebp)	No Memory abstraction	With Memory abstraction

int main(){
int a,b,d;
…..
b=a+2;
……
d=b+10;
}
	main:
	1 mov $esp,$ebp
	2 sub 24,$esp //Local Allocation		

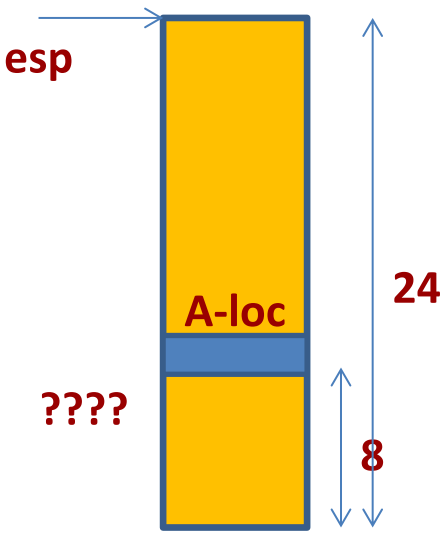
	3 mov -4(%ebp), %eax //Load a	x1	x1
	4 add $2, %eax //Compute a+2	x1+2	x1+2
	5 mov %eax, -8(%ebp) //Store b		
Symbolic Relations:	…		
	6 mov -8(%ebp), %eax //Load b	x3	x1+2
b=a+2	7 add $10, %eax //Compute b+10	x3+10	x1+12
d=a+12	8 mov %eax, -16(%ebp) //Store d		

Memory Abstraction

- Stack Memory abstraction in executables
  - Associating memory reference to a set of locations on stack.

- **Need variable like entities in executables**
  - ***A-loc*** **abstraction**: (Balakrishnan and Reps,2004)
  - Offset and size of the a-loc

Limitations to a-locs




```
main:
    sub 24, $esp
    mov $10, 8(%esp)
    call *%eax
    mov $20, 12(%esp)
```

Local Allocation

Accesses (%esp + 8)

(UNKNOWN CALL)

Accesses (%esp + 12 + ??)

StackDiff : Stack modification due to each instruction

- Not statically determinable in all scenarios

Executable artifacts

- StackDiff Indeterminable: *Unknown Calls*
  - A direct call to an external procedure with unknown prototype.
  - An indirect call with unresolved targets.
  - An indirect call and its targets have different *StackDiff.*

Existing solutions

- Existing Solutions
  - CodeSurfer/X86
    - No a-locs (variables) in such scenarios
    - Forfeits **Capability**
  - IDAPro:
    - Unsafe assumptions
    - Forfeits **Functionality**

- Our approach: Hybrid mechanism for functionality and capability
  - Static solution: Capability
  - Dynamic solution: Functionality
  - Formulate symbolic constraints based on control flow graph

Constraints Formulation

- Constraints Generation
  - *StackDiff: U*nknown stack modifications
  - *StackDepth:* Stack height at each instruction
  - Standard stack modification instructions are analyzed to derive an expression of *StackDepth* in terms of *StackDiff*

- *Boundary Conditions*
  - *StackDepth* from different paths at join points in CFG are same
    - Compiler accesses have to be deterministic
  - *StackDepth* at return instruction is zero
    - Return oriented programming

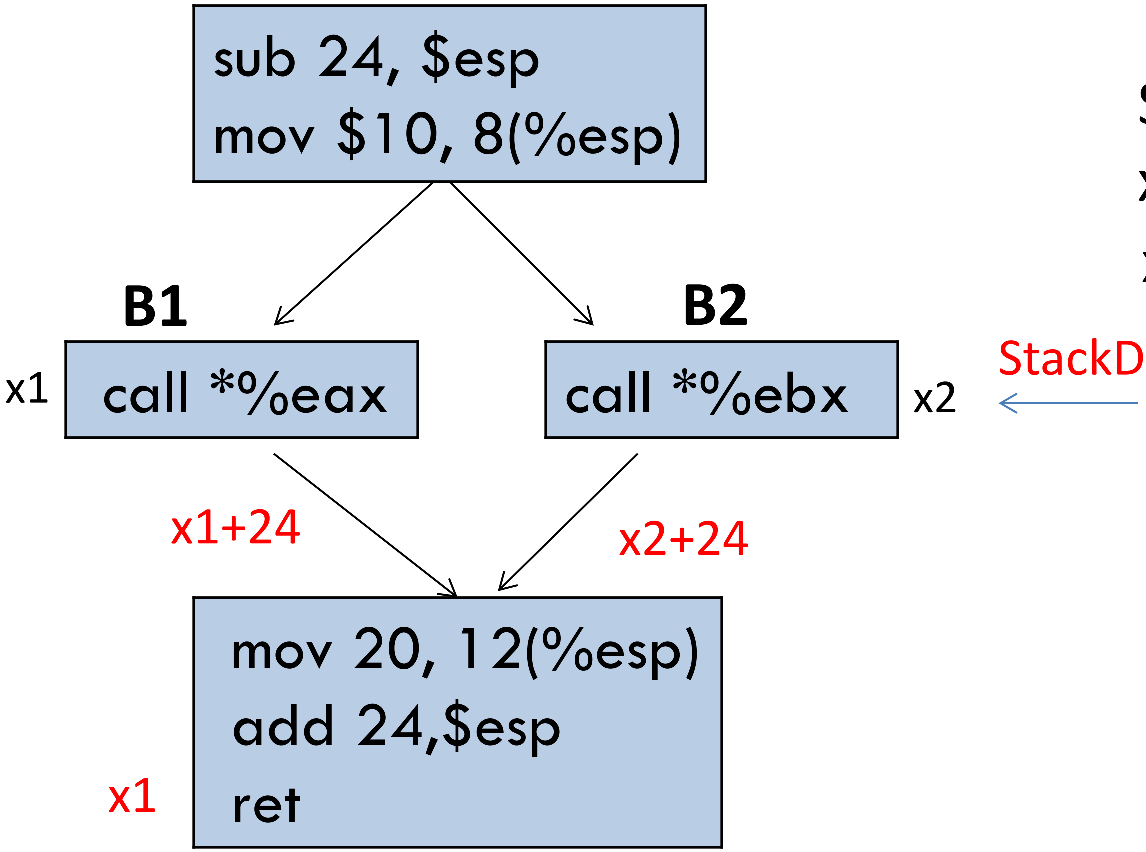
Constraints Solution

- Possible solutions
  - All *StackDiff* are available: Static solution
  - Some *StackDiff* are available
  - *None of the StackDiff* are available

Dynamic Solution

- Dynamic solution
  - Declare *StackDiff* as a return variable.
  - Dynamic adjustment of Stack Pointer in callee procedure

Example of static solution



Symbolic Equations:

x1+24 = x2+24

x1=0

Solution:

x1=0

x2=0

Example of dynamic solution

StackDiff

x1

x2

x1+x2-2

```
sub 24, %esp
mov $10, 8(%esp)
call   *%eax
call   *%ebx
sub 2,%esp
add 24,%esp
ret
```

```
sub 24, %esp
mov $10, 8(%esp)
y1=call   *%eax
sub y1,%esp
y2=call   *%ebx
sub y2,%esp
sub 2, %esp
add 24, %esp
ret
```

Symbolic Equations:

x1+x2 = 2

Static solution not possible

Symbolic Value Analysis

- SVA: A flow-sensitive context-insensitive interprocedural analysis
- Computes symbolic maps
  - Vars and a-locs
- Symbolic Grammer

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

- Symbolic Value Set: Finite set of symbolic expressions defined by the symbolic grammar
- A transfer instruction for each instruction in the IR

Symbolic Value Analysis

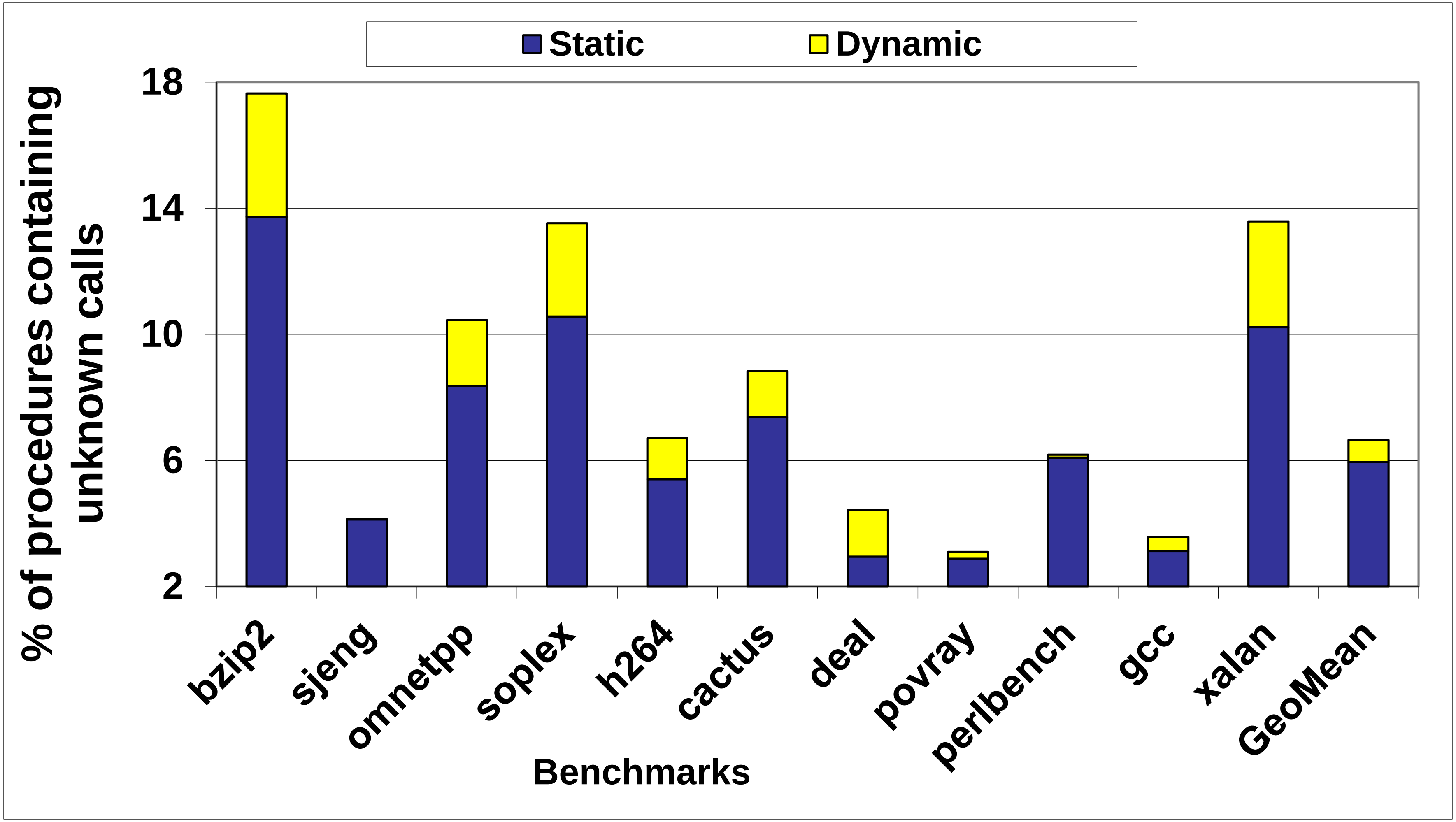
		No Memory abstraction	With Memory abstraction
	Allocations: a: -4(%ebp) b:-8(%ebp)		
int main(){	d:-16(%ebp)		
int a,b,d;			
…..	main:		
b=a+2;	1 mov $esp,$ebp		
……	2 sub 24,$esp //Local Allocation		
d=b+10;			
}	3 mov -4(%ebp), %eax //Load a	x1	x1
	4 add $2, %eax //Compute a+2	x1+2	x1+2
	5 mov %eax, -8(%ebp) //Store b		
Symbolic Relations:	…		
	6 mov -8(%ebp), %eax //Load b	x3	x1+2
b=a+2	7 add $10, %eax //Compute b+10	x3+10	x1+12
d=a+12	8 mov %eax, -16(%ebp) //Store d		

SecondWrite

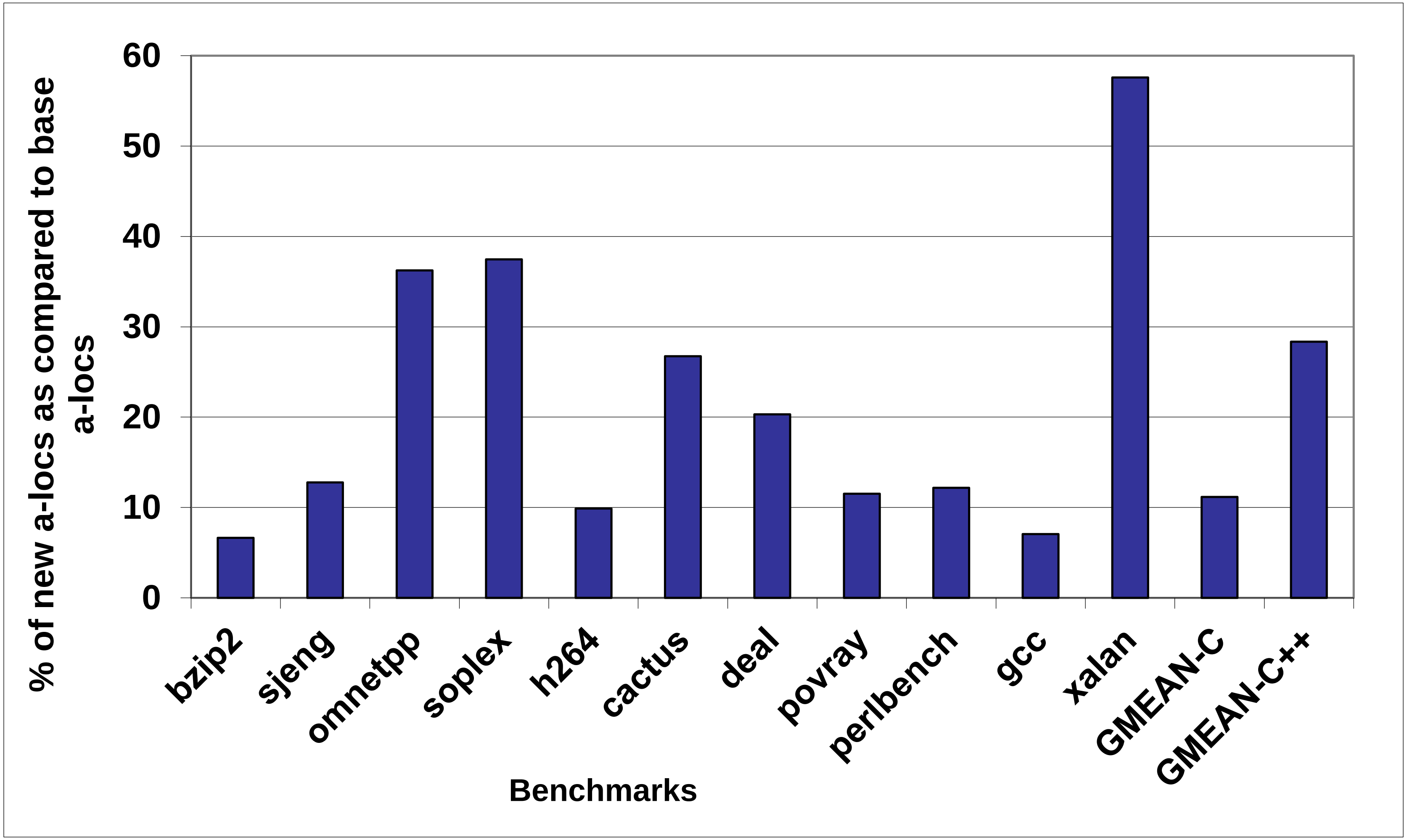
- X86 ISA support
  - Two compilers (gcc and Visual Studio)
  - Two OS (Linux and Windows)
  - Three languages (C,C++,F)
  - Spec and OMP Benchmark
  - Real programs: Apache server, Linux Coreutils

- Symbolic Framework
  - Analysis Time
    - Within 1 minute for most SPEC benchmarks
  - Storage requirements within 500 MB

Hybrid solution



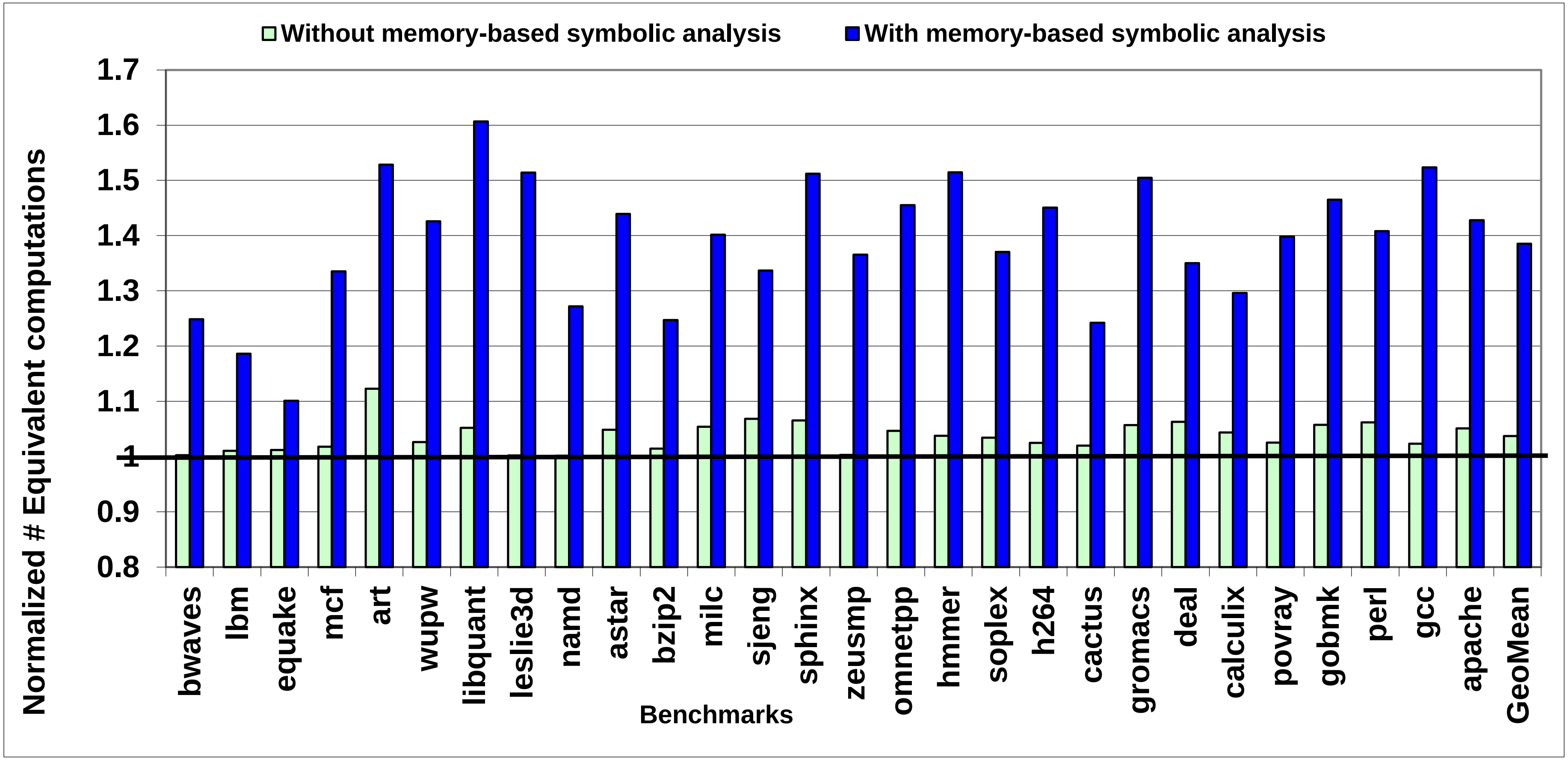
Enhanced Memory Abstraction



Applications

- Value numbering
  - Redundant instructions in optimized executables
  - Simplifies the IR, speeding up subsequent binary analysis
- Dependence Analysis
  - Parallelizing compilers employ symbolic analysis in data dependence tests
  - Extension of data dependence tests to executables
- Security Analysis
  - Extension of SVA to detect information flow vulnerabilities
  - Format string and directory traversal

Value Numbering



Security

Programs	LOC	Known/New	Type of vulnerability
Mingetty	500	N/A	N/A
Csplit	1,060	NEW	Format String
muh	2857	KNOWN	Format String
pfingerd	4689	KNOWN	Format String
gzip	5830	KNOWN	Directory Traversal
ipupdate	6,335	KNOWN	Format String
gif2png	9354	KNOWN	Directory Traversal
wu-ftpd	17576	KNOWN	Format String
tar	20518	KNOWN	Directory Traversal
KeePassX	26089	N/A	N/A
yafc	32,241	NEW	Directory Traversal
tnftp	34,762	N/A	N/A
gftp	42,390	N/A	N/A
irc2	44,837	NEW	Directory Traversal
wget	46,611	N/A	N/A
sudo	53,144	KNOWN	Format String
openssh	73335	N/A	N/A
ayttm	80,013	NEW	Format String
curl	122,248	NEW	Directory Traversal
bitchx	133,728	NEW	Format String
lynx	135876	N/A	N/A
apache	232778	N/A	N/A
mySQL	1.7 million	N/A	N/A

- Six New Vulnerabilities
- Eight Existing Vulnerabilities
- False Positive:
  - 79%
  - Source-level tools: 84%

Summary

- Improved memory abstraction model for executables
- Symbolic analysis framework for executables
- Employed the above framework for several applications

Advantages of binary frameworks

- Absence of source code
  - COTS executables and legacy binaries
  - Hand-coded assembly
- Source code analysis not reliable
  - What-You-See-Is-Not-What-You-Execute
    - Balakrishnan and Reps, 2007

Applications

- Value numbering
  - Redundant instructions in optimized executables
  - Simplifies the IR, speeding up subsequent binary analysis
- Dependence Analysis
  - Parallelizing compilers employ symbolic analysis in data dependence tests
  - Extension of data dependence tests to executables
- Security Analysis
  - Extension of SVA to detect information flow vulnerabilities
  - Format string and directory traversal

Security

Programs	LOC	Known/New	Type of vulnerability
Mingetty	500	N/A	N/A
Csplit	1,060	NEW	Format String
muh	2857	KNOWN	Format String
pfingerd	4689	KNOWN	Format String
gzip	5830	KNOWN	Directory Traversal
ipupdate	6,335	KNOWN	Format String
gif2png	9354	KNOWN	Directory Traversal
wu-ftpd	17576	KNOWN	Format String
tar	20518	KNOWN	Directory Traversal
KeePassX	26089	N/A	N/A
yafc	32,241	NEW	Directory Traversal
tnftp	34,762	N/A	N/A
gftp	42,390	N/A	N/A
irc2	44,837	NEW	Directory Traversal
wget	46,611	N/A	N/A
sudo	53,144	KNOWN	Format String
openssh	73335	N/A	N/A
ayttm	80,013	NEW	Format String
curl	122,248	NEW	Directory Traversal
bitchx	133,728	NEW	Format String
lynx	135876	N/A	N/A
apache	232778	N/A	N/A
mySQL	1.7 million	N/A	N/A

- Six New Vulnerabilities
- Eight Existing Vulnerabilities
- False Positive:
  - 79%
  - Source-level tools: 84%

Existing Tools

Property	Functional	High IR	Works without Metadata	Scalable
ATOM (Link time)	✓	X	X	✓
PLTO (Link time)	✓	X	X	✓
Spike (Link time)	✓	X	X	✓
UQBT	✓	X	X	✓
IDA Pro / Hex Rays	X	✓	✓	✓
Jakstab	X	X	✓	X
BAP (TIE)	X	✓	✓	X
CodeSurfer/X86	X	✓	✓	X
SecondWrite	✓	✓	✓	✓

Disassembly (WCRE'2013)

- Speculative disassembly
  - Apply recursive disassembly
  - Speculatively assume unknown portions as code and disassemble
  - Retain unknown portion as data in IR to maintain the correctness
  - Indirect instruction: Translate to new location in IR
- Binary characterization
  - Limits the beginning points in "unknown portions"
  - Assumption: "An address is not computed"
  - Scan text and data segment to identify all possible entry points in code segment

Transfer Functions

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	$R1 := *(R2)$	$\{F, P\} = *(\mathrm{Mem}_e(R2), s)$ $if\ \|P\| = 0$ $\quad tmp = \nabla(\bigcup_{v \in F} SM_e(v))$ $else$ $\quad tmp = \top$ $SR = \{SR - SR(R1)\} \cup \{(R1, tmp)\}$
4. Store	$*(R2) := R1$	$\{F, P\} = *(\mathrm{Mem}_e(R2), s)$ $if\ \|F\| = 1\ \&\ \|P\| = 0\ \&Func\ is\ not\ recursive\ \&$ $\quad F\ has\ no\ heap\ a\text{-}locs \quad //Strong\ Update$ $\quad SM'_e = \{\{SM_e - SM_e(v)\} \cup$ $\quad \{(v, SR(R1))\} \mid v \in F\}$ $else \quad //Weak\ Update$ $\quad SM'_e = \{\{SM_e - SM_e(y) \mid y \in \{F \cup P\}\} \cup$ $\quad \{(v, \nabla(SR(R1) \cup SM_e(v))) \mid v \in F\} \cup$ $\quad \{(p, \top) \mid p \in P\}\}$
5. SSA Phi	$R_{n+1} = \phi(R_1, R_2, \ldots, R_n)$	$SR = \{SR - SR(R_{n+1})\} \cup \{R1, \nabla(\bigcup_{i \in (1,n)} SR(R_i))\}$

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

$\mathtt{Y\_SET(F)} = \cup_{f \in F} \mathtt{StackDiff(f)}$

`BeginP` = Entry point of procedure P; $\mathtt{Pred}_{BB}$ = 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 ($I \neq \mathtt{BeginBB}$)

$S_{I'}$ = if ($I \neq \mathtt{BeginBB}$) then $S_{\mathtt{PrevI}}$ else S_{BB}

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

Initial Conditions : $S_{\mathtt{BeginP}} = 0$

Data flow rules :

For every instruction I:

I = push R $\Rightarrow$ $S_I = S_{I'} + \mathtt{size(R)}$

I = pop R $\Rightarrow$ $S_I = S_{I'} - \mathtt{size(R)}$

I = add esp, N $\Rightarrow$ $S_I = S_{I'} - N$

I = sub esp, N $\Rightarrow$ $S_I = S_{I'} + N$

I = jmp L $\Rightarrow$ $S_{\mathtt{BeginL}} = S_{I'}$

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: $\forall$ $\mathtt{Pred} \in \mathtt{Pred}_{BB}$, $S_{\mathtt{BeginBB}} = S_{\mathtt{EndPred}}$
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

Symbolic Abstraction

- Data Objects :
  - Variables
  - A-locs

- Symbolic grammar

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

- Symbolic Value Set: Finite set of symbolic expressions defined by the symbolic grammar