SVF: Static Value-Flow Analysis in LLVM

Yulei Sui, Peng Di, Ding Ye, Hua Yan and Jingling Xue

School of Computer Science and Engineering
The University of New South Wales
2052 Sydney Australia

March 18, 2016
Outline

- Static Value-Flow
- SVF Overview
- SVF Internals
- Results and Client Applications
Static Value-Flow Analysis

Statically resolves both control and data dependence of a program.

- Does the information generated at program point $A$ flow to another program point $B$ along some execution paths?
- Is there an unsafe memory access that may trigger a bug?
Static Value-Flow Analysis

Statically resolves both control and data dependence of a program.

- Does the information generated at program point $A$ flow to another program point $B$ along some execution paths?
- Is there an unsafe memory access that may trigger a bug?

Value-flow (def-use) of a variable

- Def-use of a top-level pointer (register value) is explicit on LLVM SSA.
- Def-use of an address-taken variable (allocated memory objects) is hard to compute precisely and efficiently.
Whole-Program CFG of 300.twolf (20.5KLOC)

#functions: 194  #pointers: 20773  #loads/stores: 8657

Costly to reason about flow of values on CFGs!
Call Graph of 176.gcc

#functions: 2256  #pointers: 134380  #loads/stores: 51543

Costly to reason about flow of values on CFGs!
Motivation

Why need an interprocedural static value-flow analysis in LLVM?

- **Bridge the gap between research and engineering**
  - Support developing *different analysis variants* (flow-, context-, heap-, field-sensitive analysis)
  - Minimize the efforts of implementing sophisticated analysis. *(extendable, reusable, and robust via layers of abstractions)*

Client applications:

- Static bug detection (e.g., memory leak, data-race)
- Sanitizers (e.g., MSan, TSan)

Program understanding and debugging

LLVM community support:

- Industrial-strength compiler with well-defined IR
- Front-ends that support many different languages
- Many active program analysis researchers and engineers
Motivation

Why need an interprocedural static value-flow analysis in LLVM?

- **Bridge the gap between research and engineering**
  - Support developing *different analysis variants* (flow-, context-, heap-, field-sensitive analysis)
  - Minimize the efforts of implementing sophisticated analysis. *(extendable, reusable, and robust via layers of abstractions)*

- **Client applications:**
  - *Static bug detection* (e.g., memory leak, data-race)
  - *Sanitizers* (e.g., MSan, TSan)
  - *Program understanding and debugging*
Motivation

Why need an interprocedural static value-flow analysis in LLVM?

- **Bridge the gap between research and engineering**
  - Support developing **different analysis variants** (flow-, context-, heap-, field-sensitive analysis)
  - Minimize the efforts of implementing sophisticated analysis. *(extendable, reusable, and robust via layers of abstractions)*

- **Client applications:**
  - **Static bug detection** (e.g., memory leak, data-race)
  - **Sanitizers** (e.g., MSan, TSan)
  - **Program understanding and debugging**

- **LLVM community support**
  - **Industrial-strength compiler with well-defined IR**
  - **Front-ends that support many different languages**
  - **Many active program analysis researchers and engineers**
A research tool supports refinement-based interprocedural program dependence analysis on top of LLVM.

- Pointer analysis and Value-flow construction are performed iteratively to provide increasingly improved precision for both.
- The project initially started from 2013 on LLVM-3.3. Now it supports LLVM-3.7.0 with around 50KLOC C++ code.
- Publicly available at: http://unsw-corg.github.io/SVF/
Support developing different analyses (flow-, context-, field-sensitivity)

- **Graph** is a higher-level abstraction extracted from the LLVM IR indicating *where* pointer analysis should be performed.
- **Rules** defines *how* to derive the points-to information from each statement,
- **Solver** determines in *what* order to resolve all the constraints.

More details can be found at [https://goo.gl/msaVba](https://goo.gl/msaVba).
**SVF Overview**

The SVF (Structured Value Flow) framework is a comprehensive system for the analysis of pointers and memory flow in C programs. It starts with a program in LLVM bitcode format, fed into Clang to generate LLVM IR. The system then performs various analyses to construct a value-flow graph, which is a higher-level abstraction extracted from the LLVM IR. This graph is used for context-sensitive pointer analysis, mod-ref analysis, and memory region partitioning. The value-flow graph is further refined to derive the points-to information. The framework supports different analyses such as flow-, context-, and field-sensitivity. It is designed to be flexible and can be extended to support advanced pointer analyses. The system has applications in various domains, including memory leak detection, interprocedural memory SSA construction, and pointer analysis in multithreaded programs. It is publicly available as BitVecPTAImpl, with a publicly available version at http://unsw-corg.github.io/SVF/.

**Key Components**

- **Clang**: Preprocessor and Front-End for generating LLVM IR.
- **LLVM IR**: Intermediate representation of the program.
- **BitVecPTAImpl**: Core component for bit-vector based pointer analysis.
- **Mod-Ref Analysis**: Identifies the modifiable and read-only regions.
- **Mem Region Partitioning**: Splits memory into regions based on their usage.
- **Memory SSA**: State-of-the-art SSA representation for memory variables.
- **VFG Construction**: Value-flow graph construction.
- **Solver**: Wave deep propagation and SCC topological solvers.
- **Graph Pool**: Contains constraint graph and value-flow graph.
- **Solver Pool**: Contains wave deep propagation and SCC topological solvers.
- **Points-to Data Structure**: Bit Vector for points-to analysis.
- **Conditional Points-to Map**: For regions with different versions.

**Applications**

- **Client Applications**: Include memory leak detection, interprocedural memory SSA construction, and pointer analysis in multithreaded programs.

**Tools and Techniques**

- **Interprocedural Memory SSA Construction**: Based on HSSA (CC '96).
- **Value-Flow Construction**: Dependence analysis on top of LLVM.
- **Solver**: Determines in what order to resolve all the constraints.
SVF Overview

Partition the memory objects into memory regions are accessed equivalently.
Interprocedural memory SSA construction based on HSSA (CC ’96\(^a\)) and widely used in Open64.

- **Side-Effect Annotation** at loads/stores and callsites
- **Placing Memory SSA** \(\phi\) for memory regions.
- **SSA Renaming** for regions with different versions:

\(^a\)F Chow, S Chan, SM Liu, R Lo, M Streich, *Effective representation of aliases and indirect memory operations in SSA form*, CC 1996
SVF Overview

Value-Flow Construction:

- **Direct Value-Flows**: def-use of top-level pointers
- **Indirect Value-Flows**: def-use of address-taken variables based on memory SSA
SVF Overview

- Advanced pointer analyses (SAS ’14, SPE ’14, CGO ’16)
- Memory leak detection (ISSTA ’12, TSE ’14, SAC ’16)
- Accelerating dynamic analysis including temporal memory safety (CGO ’14), spatial memory safety (ISSRE ’14, TOR ’16)
- Value-flow analysis for multithreaded programs (ICPP ’15, CGO ’16, PMAM ’16)
Outline

- Static Value-Flow
- SVF Overview
- SVF Internals
  - Pointer Analysis
  - Interprocedural Memory SSA
  - Value-Flow Construction
  - Supporting Multithreaded Programs
- Results and Client Applications
Andersen’s Pointer Analysis

SVF transforms LLVM instructions into a graph representation Constraint Graph (Design doc: https://goo.gl/Q8mxFw)

- **Node:**
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)

- **Edge:** A Constraint between two nodes
  
  **Address** $p = \text{alloc}_{obj}$
  
  **Copy** $p = q$
  
  **Load** $p = *q$
  
  **Store** $*p = q$
  
  **Field** $p = q \text{gep } f$
Andersen’s Pointer Analysis

SVF transforms LLVM instructions into a graph representation
Constraint Graph (Design doc: https://goo.gl/Q8mxFw)

- **Node:**
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)

- **Edge: A Constraint between two nodes**

Address\[p = \text{alloc}_{\text{obj}}\] \{\text{obj}\} \subseteq \text{Pts}(p)

Copy\[p = q\] \text{Pts}(p) \subseteq \text{Pts}(p)

Load\[p = \ast q\] \forall o \in \text{Pts}(q), \text{Pts}(o) \subseteq \text{Pts}(p)

Store\[\ast p = q\] \forall o \in \text{Pts}(p), \text{Pts}(q) \subseteq \text{Pts}(o)

Field\[p = q \text{ gep } f\] \forall o \in \text{Pts}(q), o.f \subseteq \text{Pts}(p)
Andersen’s Pointer Analysis

```c
struct st{
    char f1;
    char f2;
};
typedef struct st ST;

int main(){
    char a1; ST st;
    char *a = &a1;
    char *b = &(st.f2);
    swap(&a,&b);
}

void swap(char **p, char **q){
    char* t = *p;
    *p = *q;
    *q = t;
}
```

dene i32 @main() {
entry:
    %a = alloca i8*, align 8 // O1
    %b = alloca i8*, align 8 // O2
    %a1 = alloca i8, align 1 // O3
    %st = alloca %struct.st, align 1 // O4
    store i8* %a1, i8** %a, align 8
    %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
    store i8* %f2, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}

dene void @swap(i8** %p, i8** %q) {
entry:
    %0 = load i8** %p, align 8
    %1 = load i8** %q, align 8
    store i8* %1, i8** %p, align 8
    store i8* %0, i8** %q, align 8
    ret void
}
Andersen's Pointer Analysis

```c
define i32 @main() {
  entry:
    %a = alloca i8*, align 8 // O1
    %b = alloca i8*, align 8 // O2
    %a1 = alloca i8, align 1 // O3
    %st = alloca %struct.st, align 1 // O4
    store i8* %a1, i8** %a, align 8
    %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
    store i8* %f2, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
define void @swap(i8** %p, i8** %q) {
  entry:
    %0 = load i8** %p, align 8
    %1 = load i8** %q, align 8
    store i8* %1, i8** %p, align 8
    store i8* %0, i8** %q, align 8
    ret void
}
```

LLVM IR

Constraint Graph
Andersen’s Pointer Analysis

```
define i32 @main() {
  entry:
  %a = alloca i8*, align 8 // O1
  %b = alloca i8*, align 8 // O2
  %a1 = alloca i8, align 1 // O3
  %st = alloca %struct.st, align 1 // O4
  store i8* %a1, i8** %a, align 8
  %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
  store i8* %f2, i8** %b, align 8
  call void @swap(i8** %a, i8** %b)
  ret i32 0
}
define void @swap(i8** %p, i8** %q) {
  entry:
  %0 = load i8** %p, align 8
  %1 = load i8** %q, align 8
  store i8* %1, i8** %p, align 8
  store i8* %0, i8** %q, align 8
  ret void
}
```

Constraint solving techniques: Wave-Deep Propagation, HCD, LCD. More details can be found here.
Andersen’s Pointer Analysis

```llvm
define i32 @main() {
  entry:
    %a = alloca i8*, align 8 // O1
    %b = alloca i8*, align 8 // O2
    %a1 = alloca i8, align 1 // O3
    %st = alloca %struct.st, align 1 // O4
    store i8* %a1, i8** %a, align 8
    %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
    store i8* %f2, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}

define void @swap(i8** %p, i8** %q) {
  entry:
    %0 = load i8** %p, align 8
    %1 = load i8** %q, align 8
    store i8* %1, i8** %p, align 8
    store i8* %0, i8** %q, align 8
    ret void
}
```

Constraint Graph

---

EuroLLVM 2016 + CC 2016
Andersen's Pointer Analysis

```
define i32 @main() {
  entry:
    %a = alloca i8*, align 8  // O1
    %b = alloca i8*, align 8  // O2
    %a1 = alloca i8, align 1  // O3
    %st = alloca struct.st, align 1 // O4
    store i8* %a1, i8** %a, align 8
    %f2 = getElementPtr inbounds struct.st, %struct.st* %st, i32 0, i32 1
    store i8* %f2, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
define void @swap(i8** %p, i8** %q) {
  entry:
    %0 = load i8** %p, align 8
    %1 = load i8** %q, align 8
    store i8* %1, i8** %p, align 8
    store i8* %0, i8** %q, align 8
    ret void
}
```

LLVM IR

Constraint Graph
Andersen’s Pointer Analysis

```cpp
define i32 @main() {
  entry:
    %a = alloca i8*, align 8  // O1
    %b = alloca i8*, align 8  // O2
    %a1 = alloca i8, align 1  // O3
    %st = alloca %struct.st, align 1  // O4
    store i8* %a1, i8** %a, align 8
    %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
    store i8* %f2, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}

define void @swap(i8** %p, i8** %q) {
  entry:
    %0 = load i8** %p, align 8
    %1 = load i8** %q, align 8
    store i8* %1, i8** %p, align 8
    store i8* %0, i8** %q, align 8
    ret void
}
```

**LLVM IR**

**Constraint Graph**

Constraint solving techniques: Wave-Deep Propagation, HCD, LCD. More details can be found here.
Andersen’s Pointer Analysis

```c
#define i32 @main() {
entry:
  %a = alloca i8*, align 8  // O1
  %b = alloca i8*, align 8  // O2
  %a1 = alloca i8, align 1  // O3
  %st = alloca %struct.st, align 1  // O4
  store i8* %a1, i8** %a, align 8
  %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
  store i8* %f2, i8** %b, align 8
  call void @swap(i8** %a, i8** %b)
  ret i32 0
}
define void @swap(i8** %p, i8** %q) {
entry:
  %0 = load i8** %p, align 8
  %1 = load i8** %q, align 8
  store i8* %1, i8** %p, align 8
  store i8* %0, i8** %q, align 8
  ret void
}
```

LLVM IR

Constraint Graph
Andersen's Pointer Analysis

```assembly
define i32 @main() {
    %a = alloca i8*, align 8  // O1
    %b = alloca i8*, align 8  // O2
    %a1 = alloca i8, align 1  // O3
    %st = alloca struct.st, align 1  // O4
    store i8* %a1, i8** %a, align 8
    %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
    store i8* %f2, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret void
}

define void @swap(i8** %p, i8** %q) {
    %0 = load i8** %p, align 8
    %1 = load i8** %q, align 8
    store i8* %1, i8** %p, align 8
    store i8* %0, i8** %q, align 8
    ret void
}
```

Constraint solving techniques: Wave-Deep Propagation, HCD, LCD. More details can be found here.

---

EuroLLVM 2016 + CC 2016
Outline

- Static Value-Flow
- SVF Overview
- SVF Internals
  - Pointer Analysis
  - Interprocedural Memory SSA
  - Value-Flow Construction
  - Supporting Multithreaded Programs
- Results and Client Applications
Memory SSA

Memory SSA is constructed per procedure given the global points-to information after region partitioning.
Memory SSA

Memory SSA is constructed per procedure given the global points-to information after region partitioning.

- **Side-effect annotation.**
  - **Load:** $p = \ast q$ is annotated with a $\mu(o)$ for each variable $o \in \text{Pts}(q)$.
  - **Store:** $\ast p = q$ is annotated with a $o = \chi(o)$ for each variable $o \in \text{Pts}(p)$.
  - **Callsite:** $\text{foo}(\ldots)$ is annotated with $\mu(o)/\chi(o)$ if $o$ is referred or modified inside caller $\text{foo}$.
  - **Function entry/exit:** $\chi(o)/\mu(o)$ is annotated at the entry of a function (e.g., foo) if $o$ is referred or modified in $\text{foo}$. 
Memory SSA

Memory SSA is constructed per procedure given the global points-to information after region partitioning.

- **Side-effect annotation.**
  - **Load:** $p = \ast q$ is annotated with a $\mu(o)$ for each variable $o \in \text{Pts}(q)$.
  - **Store:** $\ast p = q$ is annotated with a $o = \chi(o)$ for each variable $o \in \text{Pts}(p)$.
  - **Callsite:** $\text{foo}(\ldots)$ is annotated with $\mu(o)/\chi(o)$ if $o$ is referred or modified inside caller $\text{foo}$.
  - **Function entry/exit:** $\chi(o)/\mu(o)$ is annotated at the entry of a function (e.g., $\text{foo}$) if $o$ is referred or modified in $\text{foo}$.

- **Memory SSA construction**
  - **Placing Memory SSA $\phi$** for memory objects.
  - **Renaming** objects with different versions:
    - $\mu(o)$ is treated as a use of $o$.
    - $o = \chi(o)$ is treated as both a def and a use of $o$. 
define i32 @main() {
  entry:
    %a = alloca i8*, align 8  // O1
    %b = alloca i8*, align 8  // O2
    %a1 = alloca i8, align 1  // O3
    %st = alloca %struct.st, align 1  // O4
    store i8* %a1, i8** %a, align 8
    %f2 = getelementptr ... %st, i32 0, i32 1
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
define void @swap(i8** %a, i8** %b) {
  entry:
    %0 = load i8** %p, i8** %q, align 8
    %1 = load i8** %q, align 8
    store i8* %1, i8** %p, align 8
    store i8* %0, i8** %q, align 8
    ret void
}
Memory SSA

=====FUNCTION: main=====

entry
  %a = alloca i8*, align 8 // O1
  %b = alloca i8*, align 8 // O2
  %a1 = alloca i8, align 1 // O3
  %st = alloca %struct.st, align 1 // O4

store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)

%f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)

CALMU(MR1V_2)
CALMU(MR2V_2)
call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)

ret i32 0

=====FUNCTION: swap=====

MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
%0 = load i8*, i8** %p, align 8
LDMU(MR2V_1)
%1 = load i8*, i8** %q, align 8

store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)

store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)

ret void
RETMU(MR1V_2)
RETMU(MR2V_2)

Pre-computed Points-to:
pt(%a) = pt(%p) = {O1}
pt(%b) = pt(%q) = {O2}

Memory Region:
MR2: O2
MR1: O1

Annotated CHIs at stores
Memory SSA

```
========FUNCTION: main========
entry
  %a = alloca i8*, align 8  // O1
  %b = alloca i8*, align 8  // O2
  %a1 = alloca i8, align 1  // O3
  %st = alloca %struct.st, align 1  // O4
  store i8* %a, i8** %p, align 8
  MR1V_2 = STCHI(MR1V_1)
  %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
  store i8* %f2, i8** %b, align 8  %st = alloca %struct.st, align 1
  MR2V_2 = STCHI(MR2V_1)
  CALMU(MR1V_2)
  CALMU(MR2V_2)
  call void @swap(i8** %a, i8** %b)
  MR1V_3 = CALCHI(MR1V_2)
  MR2V_3 = CALCHI(MR2V_2)
  ret i32 0

========FUNCTION: swap========
entry
  %st = alloca %struct.st, i8* %q, %struct.st* %st, i32 0, i32 1
  store i8* %0, i8** %q, align 8
  MR1V_1 = ENCHI(MR1V_0)
  MR2V_1 = ENCHI(MR2V_0)
  LDMU(MR1V_1)
  %0 = load i8*, i8** %p, align 8
  LDMU(MR2V_1)
  %1 = load i8*, i8** %q, align 8
  store i8* %1, i8** %p, align 8
  MR1V_2 = STCHI(MR1V_1)
  store i8* %0, i8** %q, align 8
  MR2V_2 = STCHI(MR2V_1)
  ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Pre-computed Points-to:
- pt(%a) = pt(%p) = {O1}
- pt(%b) = pt(%q) = {O2}

Memory Region:
- MR2: O2
- MR1: O1

Annotated MUs/CHIs at call sites:
- Function entry/exit
- Function entry/exit

Annotated MUs at loads:
- LDMU(MR1V_1)
- LDMU(MR2V_1)
- store i8* %0, i8** %q, align 8
- store i8* %f2, i8** %b, align 8
- store i8* %a1, i8** %a, align 8
- store i8* %1, i8** %q, align 8
- REBMU(MR1V_2)
- RETMU(MR2V_2)
Memory SSA

Pre-computed Points-to:

\[
\begin{align*}
\text{pt}(\%a) &= \text{pt}(\%p) = \{O1\} \\
\text{pt}(\%b) &= \text{pt}(\%q) = \{O2\}
\end{align*}
\]

Memory Region:

\[
\begin{align*}
\text{MR2: } O2 \\
\text{MR1: } O1
\end{align*}
\]

Annotated MUs/CHIs at callsite
Pre-computed Points-to:
pt(%a) = pt(%p) = \{O1\}
pt(%b) = pt(%q) = \{O2\}

Memory Region:
MR2: O2
MR1: O1

Annotated MUs/CHIs at Function entry/exit
package Memory SSA

```
1  define i32 @main() {
2    entry:
3      %a = alloca i8*, align 8 // O1
4      %b = alloca i8*, align 8 // O2
5      %a1 = alloca i8, align 1 // O3
6      %st = alloca %struct.st, align 1 // O4
7      store i8* %a, i8** %st, align 8 // O1
8      %f2 = getelementptr ... %st, i32 0, i32 1 // O4
9      store i8* %f2, i8** %b, align 8 // O2
10     call void @swap(i8** %a, i8** %b) // O1
11     ret i32 0
12  }
13  define void @swap(i8** %p, i8** %q) {
14    entry:
15      %0 = load i8** %p, align 8 // O1
16      %1 = load i8** %q, align 8 // O2
17      store i8* %1, i8** %p, align 8 // O1
18      store i8* %0, i8** %q, align 8 // O2
19      ret void
20  }
```

```
# LLVM IR

entry:
  %a = alloca i8*, align 8 // O1
  %b = alloca i8*, align 8 // O2
  %a1 = alloca i8, align 1 // O3
  %st = alloca %struct.st, align 1 // O4
  store i8* %a, i8** %st, align 8 // O1
  %f2 = getelementptr ... %st, i32 0, i32 1 // O4
  store i8* %f2, i8** %b, align 8 // O2
  call void @swap(i8** %a, i8** %b) // O1
  ret i32 0
```
Outline

- Static Value-Flow
- SVF Overview
- SVF Internals
  - Pointer Analysis
  - Interprocedural Memory SSA
  - Value-Flow Construction
  - Supporting Multithreaded Programs
- Results and Client Applications
Interprocedural Value-Flow

---

**FUNCTION: main**

entry

```
%a = alloca i8*, align 8  // O1
%b = alloca i8*, align 8  // O2
%a1 = alloca i8, align 1   // O3
%st = alloca %struct.st, align 1 // O4
```

```
store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)
%f2 = getelementptr ... %st, ...
store i8* %f2, i8** %b, align 8
```

---

**FUNCTION: swap**

```
%a1 = alloca
```

```
%1 = load i8*, i8** %b
ret i32 0
```

---

**Annotated IR**

**Value-Flow Graph**
```c
struct st{
    char f1;
    char f2;
};
typedef struct st ST;

int main()
{
    char a1;
    ST st;
    char *a = &a1;
    char *b = &(st.f2);
    swap(&a, &b);
}

void swap(char **p, char **q)
{
    char *t = *p;
    *p = *q;
    *q = t;
}
```

Annotated IR Value-Flow Graph

---

Interprocedural Value-Flow

---

EuroLLVM 2016 + CC 2016
struct st { char f1; char f2; }
typedef struct st ST;

int main()
{
    char a1;
    ST st;
    char *a = &a1;
    char *b = &(st.f2);
    swap(&a, &b);
}

void swap(char **p, char **q)
{
    char *t = *p;
    *p = *q;
    *q = t;
}

Annotated IR

Value-Flow Graph

Interprocedural Value-Flow

EuroLLVM 2016 + CC 2016
void swap(char **p, char **q) {
    char* t = *p;
    *p = *q;
    *q = t;
}

int main() {
    char a1;
    ST st;
    char *a = &a1;
    char *b = &(st.f2);
    swap(&a, &b);
}

struct st {
    char f1;
    char f2;
};
typedef struct st ST;

Annotated IR

Value-Flow Graph
struct st { char f1; char f2; };
typedef struct st ST;
int main(){
char a1;
ST st;
char *a = &a1;
char *b = &(st.f2);
swap(&a, &b);
}

void swap(char **p, char **q){
char *t = *p;
*p = *q;
*q = t;
}

int main();
Interprocedural Value-Flow

entry
%a = alloca i8*, align 8 // O1
%b = alloca i8*, align 8 // O2
%a1 = alloca i8, align 1 // O3
%st = alloca %struct.st, align 1 // O4
store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)
%f2 = getelementptr ... %st, ...
store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)
CALMU(MR1V_2)
call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)
LDMU(MR2V_3)
%0 = load i8*, i8** %b
ret i32 0

entry
%a = alloca i8*, align 8         // O1
%b = alloca i8*, align 8         // ... %st
store %f2, %b
2V_2 = STCHI(MR2V_1)
swap
main
%a1%a1 %f2%f2
%st%st
Annotated IR Value-Flow Graph
Interprocedural Value-Flow

```c
struct st{
    char f1;
    char f2;
};
typedef struct st ST;

int main(){
    char a1;
    ST st;
    char *a = &a1;
    char *b = &(st.f2);
    swap(&a,&b);
}

void swap(char **p, char **q){
    char* t = *p;
    *p = *q;
    *q = t;
}
```

```c
define i32 @main() {
    entry:
        %a = alloca i8*, align 8 // O1
        %b = alloca i8*, align 8 // O2
        %a1 = alloca i8, align 1 // O3
        %st = alloca %struct.st, align 1 // O4
        store i8* %a1, i8** %a, align 8
        %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
        store i8* %f2, i8** %b, align 8
        call void @swap(i8** %a, i8** %b)
        ret i32 0
}

define void @swap(i8** %p, i8** %q) {
    entry:
        %0 = load i8** %p, align 8
        %1 = load i8** %q, align 8
        store i8* %1, i8** %p, align 8
        store i8* %0, i8** %q, align 8
        ret void
}
```

Query: b ?

```
Usage: python -m venv env
source env/bin/activate
pip install -r requirements.txt
python -m pip install -e .
python main.py
```
struct st { char f1; char f2; }
typedef struct st ST;

int main() { 
char a1; ST st; 
char *a = &a1; 
char *b = &(st.f2); 
swap(&a, &b); 
}

void swap(char ** p, char ** q) {
char * t = *p; 
*p = *q; 
*q = t;
}

EuroLLVM 2016 + CC 2016

Annotated IR

Value-Flow Graph

Interprocedural Value-Flow
```c
struct st { char f1; char f2; }
typedef struct st ST;

int main() {
    char a1; ST st;
    char *a = &a1;
    char *b = &(st.f2);
    swap(&a, &b);
}

void swap(char **p, char **q) {
    char *t = *p;
    *p = *q;
    *q = t;
}
```

Annotated IR

Value-Flow Graph

---

**Interprocedural Value-Flow**

---

@EuroLLVM 2016 + CC 2016
struct st { char f1; char f2; }
typedef struct st ST;
int main() {
    char a1;
    ST st;
    char *a = &a1;
    char *b = &(st.f2);
    swap(&a, &b);
}

void swap(char **p, char **q) {
    char *t = *p;
    *p = *q;
    *q = t;
}

Annotated IR

Value-Flow Graph

EuroLLVM 2016 + CC 2016
Outline

- Static Value-Flow
- SVF Overview
- SVF Internals
  - Pointer Analysis
  - Interprocedural Memory SSA
  - Value-Flow Construction
  - Supporting Multithreaded Programs
- Results and Client Applications
Value-Flow Under Thread Interleaving

Thread 1

s₁ : *p = & b
s₂ : *p = & c
s₃ : q = *p

(a) non-interference via join
join(t₂)

(b) non-interference via lock/unlock
lock(l)
unlock(l)
lock(l)
unlock(l)

fork(t₂, foo)
    s₂ : *p = & c
   s₃  :  q =  *p
Thread 1
     foo(){
    s₁ : *p = & b
}

(a) ... 2
     foo(){
    s₁ : *p = & b
}
(b) non-interference via lock/unlock
lock(l)
unlock(l)
lock(l)
unlock(l)
Value-Flow Under Thread Interleaving

Thread 1

fork(t2, foo)

s1: *p = & b

s2: *p = & c

s3: q = *p

Thread 2

foo(){

s1: *p = & b

}

Interleaving

(a) non-interference via join

join(t2)

(b) non-interference via lock/unlock

lock(l)

unlock(l)

lock(l)

unlock(l)
Value-Flow Under Thread Interleaving

Scenario 1:

Thread 1

- s1: *p = & b
- s2: *p = & c
- s3: q = *p

Thread 2

- s1: *p = & b

execution sequence : s1, s2, s3
Value-Flow Under Thread Interleaving

Scenario 2:

- **Thread 1**
  - `s1 : *p = & b`
  - `s2 : *p = & c`
  - `s3 : q = *p`

- **Thread 2**
  - `s1 : *p = & b`

**Execution sequence:**
- Scenario 1: `s1, s2, s3`
- Scenario 2: `s2, s1, s3`

(a) non-interference via join
(b) non-interference via lock/unlock
Value-Flow Under Thread Interleaving

Scenario 3:

Thread 1

s2 : *p = & c
s3 : q = *p

Thread 2

s1 : *p = & b

execution sequence : s1, s2, s3
execution sequence : s2, s1, s3
execution sequence : s2, s3, s1
Value-Flow Under Thread Interleaving

Thread 1

fork(t2, foo)
join(t2)

s₁ : *p = & b
s₂ : *p = & c
s₃ : q = *p

(a) non-interference via join

Thread 2

foo(){
}

s₂ : *p = & c
s₃ : q = *p

Thread 1

fork(t2, foo)

lock(l)

s₁ : *p = & b

Thread 2

foo(){
}

s₂ : *p = & c
s₃ : q = *p

unlock(l)

(b) non-interference via lock/unlock

Scenario 1:
execution sequence : s₁, s₂, s₃

Scenario 2:
execution sequence : s₂, s₁, s₃

Scenario 3:
execution sequence : s₂, s₃, s₁
Supporting Multithreaded Programs \textbf{(CGO ’16)}

- Thread-oblivious value-flows (Ignore Pthread APIs)
- Thread-aware value-flows
  - Fork-join
  - Memory access
  - Lock/unlock
Supporting Multithreaded Programs (CGO ’16)

- Thread-oblivious value-flows (Ignore Pthread APIs)
- Thread-aware value-flows
  - Fork-join
  - Memory access
  - Lock/unlock
Supporting Multithreaded Programs (CGO ’16)

- Thread-oblivious value-flows (Ignore Pthread APIs)
- Thread-aware value-flows
  - Fork-join
  - Memory access
  - Lock/unlock

Pre-analysis

- fork/join

Interleaving analysis

- MHP pairs

Value-flow analysis

- aliased pairs

Lock analysis

- thread-aware def-use

+ thread-oblivious def-use

final value-flows
Supporting Multithreaded Programs (CGO ’16)

- Thread-oblivious value-flows (Ignore Pthread APIs)
- Thread-aware value-flows
  - Fork-join
  - Memory access
  - Lock/unlock
Supporting Multithreaded Programs (CGO ’16)

- Thread-oblivious value-flows (Ignore Pthread APIs)
- Thread-aware value-flows
  - Fork-join
  - Memory access
  - Lock/unlock
Outline

- Static Value-Flow
- SVF Overview
- SVF Internals
  - Pointer Analysis
  - Interprocedural Memory SSA
  - Value-Flow Construction
  - Supporting Multithreaded Programs
- Results and Client Applications
Evaluation and Results

- **Benchmarks:**
  - All SPEC C benchmarks: 15 programs from CPU2000 and 12 programs from CPU2006
  - 20 Open-source applications: most of them are recent released versions.
  - Total lines of code evaluated: 8,005,872 LOC with maximum program size 2,720,279 LOC
  - Gold Plugin is used to combine multiple bitcode files into one

- **Machine setup:**
  - Ubuntu Linux 3.11.0-15-generic Intel Xeon Quad Core HT, 3.7GHZ, 64GB
Analysis Time

Total Analysis Time = Andersen + MemorySSA + VFG
Analysis Time Distribution

Average Percentage: Andersen (71.9%), Memory SSA (11.3%), VFG (16.8%)
Analysis Time: Andersen v.s. Flow-Sensitive Points-to Analysis

Flow-Sensitive Analysis Slowdowns: From 1.2× to 44×. On average 6.5×.
Client 1: Memory Leak Detection (TSE ’14, ISSTA ’12)

- **Source-Sink Problem**: every object created at an allocation site (a source) must eventually reach one free site (a sink) during any program execution path.

```
  p = malloc
  free(p) free(p)
  Leak path
```

```
Client 1: Memory Leak Detection (TSE ’14, ISSTA ’12)

- **Source-Sink Problem**: every object created at an allocation site (a source) must eventually reach one free site (a sink) during any program execution path.

```
  p = malloc
  free(p) free(p)
  Leak path
```
Client 1: Memory Leak Detection (TSE ’14, ISSTA ’12)

<table>
<thead>
<tr>
<th>Leak Detector</th>
<th>Speed (LOC/sec)</th>
<th>Bug Count</th>
<th>FP Rate(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATHENA</td>
<td>50</td>
<td>53</td>
<td>10</td>
</tr>
<tr>
<td>CONTRADICTION</td>
<td>300</td>
<td>26</td>
<td>56</td>
</tr>
<tr>
<td>CLANG</td>
<td>400</td>
<td>27</td>
<td>25</td>
</tr>
<tr>
<td>SPARROW</td>
<td>720</td>
<td>81</td>
<td>16</td>
</tr>
<tr>
<td>FASTCHECK</td>
<td>37,900</td>
<td>59</td>
<td>14</td>
</tr>
<tr>
<td>SABER</td>
<td>10,220</td>
<td>85</td>
<td>19</td>
</tr>
</tbody>
</table>

Comparing SABER with other static detectors on analysing SPEC2000 C programs
Client 2: Accelerating Memory Sanitizer (*CGO ’14*)

- Detecting uninitialized variables using source-level instrumentation on LLVM IR\(^1\).

\[\text{def} \rightarrow \text{check point} \rightarrow \text{def} \rightarrow \text{check point} \rightarrow \text{def} \rightarrow \text{undefined} \]

\[\begin{align*}
\text{Safe} & \quad \text{No Check} \\
\text{Maybe unsafe} & \quad \text{Needs Check}
\end{align*}\]

There is no value-flow reachable from an undefined allocation sites

Reachable from an undefined allocation sites along at least one value-flow path

\(^1\text{In C, global variables are default-initialized but local and heap variables are not}\)
Client 2: Accelerating Memory Sanitizer (CGO ’14)

- **USHER\(^{TL}\)**: Direct value-flow via only top-level pointers.
- **USHER\(^{AT}\)**: Direct and indirect value-flow via both top-level and address-taken variables.
- **USHER\(^{Op1}\)**: Reduce shadow propagation. \(X \rightarrow Y \rightarrow Z\) reduced to \(X \rightarrow Z\).
- **USHER**: Redundant check elimination. Check \(c\) is redundant if all checks flow to \(c\) is checked (e.g., double checking \(*p\) along control flow).
Client 2: Accelerating Memory Sanitizer (CGO ’14)

- **USHER\textsuperscript{TL}:** Direct value-flow via only top-level pointers.
- **USHER\textsuperscript{AT}:** Direct and indirect value-flow via both top-level and address-taken variables.
- **USHER\textsuperscript{Op1}:** Reduce shadow propagation. $X \rightarrow Y \rightarrow Z$ reduced to $X \rightarrow Z$.
- **USHER:** Redundant check elimination. Check $c$ is redundant if all checks flow to $c$ is checked (e.g., double checking $*p$ along control flow).

### Execution time slowdowns (normalized with respect to native code).

![Graph showing execution time slowdowns]
Client 3: Accelerating Thread Sanitizer (PMAM ’16)

A check for a memory access is redundant if it has no outgoing or incoming inter-thread (thread-aware) value-flows.

- ThreadLocal
- MHP analysis + Alias analysis + Lock Analysis

Speedups over original TSan (under -O0).
Conclusion and Future Work

• Conclusion
  • A research tool supports refinement-based interprocedural program dependence analysis on top of LLVM
  • Pointer analysis variants
  • Interprocedural memory SSA
  • Value-flow construction

• Future work
  • Advanced Static Analysis
    • C++/Objective-C Support
    • Demand-driven flow-, context- and heap-sensitive analysis (coming soon)
  • Bug Detection
    • Use-after-free detection
    • Enforcement of fine-grained control flow integrity (CFI)
    • Accelerating dynamic symbolic execution (Klee or Fuzzer etc.)
• Publicly available open source of SVF (LLVM License)  
  http://unsw-corg.github.io/SVF/
• Micro benchmarks to validate the correctness of analyses  
  https://github.com/unsw-corg/PTABen
• A simple Eclipse Plugin  
  https://github.com/unsw-corg/SVF-EclipsePlugin

Thanks!

Q & A