Accelerating Dynamic Detection of Uses of Undefined Values with Static Value-Flow Analysis

Ding Ye Yulei Sui Jingling Xue
Programming Languages and Compilers Group
School of Computer Science and Engineering
University of New South Wales, NSW 2052, Australia
{dye, ysui, jingling}@cse.unsw.edu.au

ABSTRACT

Uninitialized variables can cause system crashes when used and security vulnerabilities when exploited. With source rather than binary instrumentation, dynamic analysis tools such as MSan can detect uninitialized memory uses at significantly reduced overhead but are still costly.

In this paper, we introduce a static value-flow analysis, called Usher, to guide and accelerate the dynamic analysis performed by such tools. Usher reasons about the definedness of values using a value-flow graph (VFG) that captures def-use chains for both top-level and address-taken variables interprocedurally and removes unnecessary instrumentation by solving a graph reachability problem. Usher works well with any pointer analysis (done a priori) and facilitates advanced instrumentation-reducing optimizations (with two demonstrated here). Implemented in LLVM and evaluated using all the 15 SPEC2000 C programs, Usher can reduce the slowdown of MSan from 212% – 302% to 123% – 140% for a number of configurations tested.

Categories and Subject Descriptors

F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms, languages, reliability, performance

Keywords

Undefined values, static and dynamic analysis

1. INTRODUCTION

Uninitialized variables in C/C++ programs can cause system crashes if they are used in some critical operations (e.g., pointer dereferencing and branches) and security vulnerabilities if their contents are controlled by attackers. The undefinedness of a value can be propagated widely throughout a program directly (via assignments) or indirectly (via the results of operations using the value), making uses of undefined values hard to detect efficiently and precisely.

Static analysis tools [3, 14] can warn for the presence of uninitialized variables but usually suffer from a high false positive rate. As such, they typically sacrifice soundness (by missing bugs) for scalability in order to reduce excessively high false positives that would otherwise be reported.

To detect more precisely uses of undefined values (with fairly low false positives), dynamic analysis tools are often used in practice. During an instrumented program's execution, every value is shadowed, and accordingly, every statement is also shadowed. For a value, its shadow value maintains its definedness to enable a runtime check to be performed on its use at a *critical operation* (Definition 1).

The instrumentation code for a program can be inserted into either its binary [2, 24] or its source [9, 12]. *Binary instrumentation* causes an order of magnitude slowdown (typically 10X - 20X). In contrast, *source instrumentation* can be significantly faster as it reaps the benefits of optimizations performed at compile time. For example, MSan (MemorySanitizer) [9], a state-of-the-art tool that adopts the latter approach, is reported to exhibit a typical slowdown of 3X but is still costly, especially for some programs.

Both approaches suffer from the problem of blindly performing shadow propagations for all the values and definedness checks at all the critical operations in a program. In practice, most values in real programs are defined. The shadow propagations and checks on a large percentage of these values can be eliminated since their definedness can be proved statically. In addition, a value that is never used at any critical operation does not need to be tracked.

In this paper, we present a static value-flow analysis framework, called Usher, to accelerate uninitialized variable detection performed by source-level instrumentation tools such as MSan for C programs. We demonstrate its usefulness by evaluating an implementation in LLVM against MSan using all the 15 SPEC2000 C programs. Specifically, this paper makes the following contributions:

- We introduce a new static value-flow analysis, Usher, for detecting uses of undefined values in C programs. Usher reasons about statically the definedness of values using a value-flow graph (VFG) that captures def-use chains for all variables interprocedurally and removes unnecessary instrumentation by solving a graph reachability problem. Usher is field-, flow- and context-sensitive wherever appropriate and supports

CGO '14, February 15 - 19 2014, Orlando, FL, USA

ACM 978-1-4503-2670-4/14/02 ...$15.00.

two flavors of strong updates. Our value-flow analysis is sound (by missing no bugs statically) as long as the underlying pointer analysis is. This work represents the first such whole-program analysis for handling top-level and address-taken variables to guide dynamic instrumentation for C programs.

- We show that our VFG representation allows advanced instrumentation-reducing optimizations to be developed (with two demonstrated in this paper). In addition, its precision can be improved orthogonally by leveraging existing and future advances on pointer analysis.
- We show that USHER, which is implemented in LLVM, can reduce the slowdown of MSAN from 212% – 302% to 123% – 140% for all the 15 SPEC2000 C programs under a number of configurations tested.

The rest of the paper is organized as follows. Section 2 introduces a subset of C as the basis to present our techniques. Section 3 describes our USHER framework. Section 4 presents and analyzes our experimental results. Section 5 discusses the related work. Section 6 concludes.

2. PRELIMINARIES

In Section 2.1, we introduce TINYC, a subset of C, to allow us to present our USHER framework succinctly. In Section 2.2, we highlight the performance penalties incurred by shadow-memory-based instrumentation.

2.1 TINYC

As shown in Figure 1, TINYC represents a subset of C. A program is a set of functions, with each comprising a list of statements (marked by labels from *Lab*) followed by a return. TINYC includes all kinds of statements that are sufficient to present our techniques: assignments, memory allocations, loads, stores, branches and calls. We distinguish two types of allocation statements, (1) $x := \text{alloc}_\rho^{\mathcal{T}}$, where the allocated memory ρ is initialized, and (2) $x := \text{alloc}_\rho^{\mathcal{F}}$, where the allocated memory ρ is not initialized.

Without loss of generality, we consider only local variables, which are divided into (1) the set Var^{TL} of *top-level variables* (accessed directly) and (2) the set Var^{AT} of *address-taken variables* (accessed indirectly only via top-level pointers). In addition, all variables in $Var^{TL} \cup Var^{AT}$ and all constants in *Const* have the same type.

TINYC mimics LLVM-IR [15] in how the & (address) operation as well as loads and stores are represented. In TINYC, as illustrated in Figure 2, & is absent since the addresses of variables are taken by using $\text{alloc}_\rho^{\mathcal{T}}$ and $\text{alloc}_\rho^{\mathcal{F}}$ operations and the two operands at a load/store must be both top-level variables. In Figure 2(c), we have $Var^{TL} = \{\texttt{a}, \texttt{i}, \texttt{x}, \texttt{y}\}$, $Var^{AT} = \{\texttt{b}, \texttt{c}\}$ and $Const = \{\texttt{10}\}$.

2.2 Shadow-Memory-based Instrumentation

When a program is fully instrumented with shadow memory [2, 9, 12, 24], the definedness of every variable v in $Var^{TL} \cup Var^{AT}$ is tracked by its shadow variable, $\underline{v} \in \{\mathcal{T}, \mathcal{F}\}$, of a Boolean type. All constant values in *Const* are defined (with $\mathcal{T}$). Whether a variable is initialized with a defined value or not upon declaration depends on the default initialization rules given. In C programs, for example, global variables are default-initialized but local variables are not.

P	::=	F^+	(program)
F	::=	def $f(a)$ $\{\overline{\ell : stmt;}$ ret $r;\}$	(function)
$stmt$	::=	$x := n$	(constant copy)
	|	$x := y$	(variable copy)
	|	$x := y \otimes z$	(binary operation)
	|	$x := \text{alloc}_\rho^{\mathcal{T}}$	(allocation with ρ initialized)
	|	$x := \text{alloc}_\rho^{\mathcal{F}}$	(allocation with ρ not initialized)
	|	$x := *y$	(load)
	|	$*x := y$	(store)
	|	$x := f(y)$	(call)
	|	if x goto ℓ	(branch)

$x, y, z, a, r \in Var^{TL}$ $\quad \rho \in Var^{AT}$ $\quad n \in Const$ $\quad \ell \in Lab$

Figure 1: The TINYC source language.

(a) C	(b) LLVM	(c) TINYC
`int **a, *b;`	`a = alloc_b`	$\texttt{a} := \text{alloc}_\texttt{b}^{\mathcal{F}}$;
`int c, i;`	`x = alloc_c`	$\texttt{x} := \text{alloc}_\texttt{c}^{\mathcal{F}}$;
`a = &b;`	`STORE x, a`	$*\texttt{a} := \texttt{x}$;
`b = &c;`	`STORE 10, x`	$\texttt{y} := 10$;
`c = 10;`	`i = LOAD x`	$*\texttt{x} := \texttt{y}$;
`i = c;`		$\texttt{i} := *\texttt{x}$;

Figure 2: The TINYC representation vs. LLVM-IR (where x and y are top-level temporaries).

As the results produced by statements may be tainted by the undefined values used, every statement s is also instrumented by its shadow, denoted $\underline{s}$. For example, $x := y \otimes z$ is instrumented by $\underline{x := y \otimes z}$, which implies that $\underline{x} := \underline{y} \wedge \underline{z}$ is executed at run time to enable shadow propagations, where $\wedge$ represents the Boolean *AND* operator.

DEFINITION 1 (CRITICAL OPERATIONS). An operation performed at a load, store or branch is a *critical operation*.

A runtime check is made for the use of a value at every critical operation. If its shadow is $\mathcal{F}$, a warning is issued.

By indiscriminately tracking all values and propagating their shadow values across all statements in a program, *full instrumentation* can slow the program down significantly.

3. METHODOLOGY

As shown in Figure 3, USHER, which is implemented in LLVM, comprises five phases (described below). In "Memory SSA Construction", each function in a program is put in SSA (Static Single Assignment) form based on the pointer information available. In "Building VFG", a VFG that connects def-use chains interprocedurally is built (flow-sensitively) with two flavors of strong updates being supported. In "Definedness Resolution", the definedness of all values is statically resolved context-sensitively. In "Guided Instrumentation", the instrumentation code required is generated, with strong updates performed to shadow values. This phase is regarded as the key contribution of this paper. In "VFG-based Optimizations", some VFG-based optimizations are applied to reduce instrumentation overhead further. Compared to full instrumentation, our guided instrumentation is more lightweight.

USHER is sound as long as the underlying pointer analysis is. So no uses of undefined values will be missed. In addition to being flow- and context-sensitive, our value-flow analysis is also field-sensitive to obtain improved precision.

3.1 Memory SSA Construction

Initially, USHER puts all functions in a program in SSA form, an IR where a variable is statically defined exactly once. In TINYC (as in LLVM-IR), def-use information for top-level variables is immediately available. However, def-use information for address-taken variables requires pointer analysis to discover how they are accessed indirectly as *indirect defs* at stores and *indirect uses* at loads.

Figure 4 shows how TINYC is extended to allow a TINYC program to be put in SSA form. Note that ϕ is the standard function for handling control-flow join points. Following [6], we use μ and χ functions to, respectively, indicate the potentially indirect uses and defs of address-taken variables at loads, stores and allocation sites. Each load $x := *y$ is annotated with a list $\overline{\mu(\rho)}$ of μ functions, where each $\mu(\rho^k)$ function represents potentially an indirect use of ρ^k (that may be pointed to by y). Similarly, each store $*x := y$ is annotated with a list $\overline{\rho := \chi(\rho)}$ of χ functions, where each $\rho^k := \chi(\rho^k)$ function represents potentially an indirect use and def of ρ^k (that may be pointed to by x). At an allocation site, a single $\rho := \chi(\rho)$ function is added, where ρ is the name of the address-taken variable allocated.

A function def $f(a)$ $\{..., \text{ret } r;\}$ is extended to make explicit (1) all address-taken variables (called *virtual formal parameters*) that are used, i.e., read in f directly or indirectly via a, and (2) all address-taken variables (called *virtual output parameters*) that are either modified in f via a or returned by r, directly or indirectly. Accordingly, the syntax for the call sites of f is extended. For a function f and its call sites, ρ^k (the k-th element) in each of the $\overline{\rho}$ lists used always represents the same address-taken variable.

Once all required μ and χ functions have been added, every function is put in SSA form individually by using a standard SSA construction algorithm. Figure 5 gives an example. It is understood that different occurrences of a variable with the same version (e.g., $\mathtt{b}_1$ and $\mathtt{b}_2$) are different if they appear in different functions. Recall that each $\rho := \chi(\rho)$ function represents a potential use and def of ρ [6]. In $\mathtt{b}_2 := \chi(\mathtt{b}_1)$ associated with $*\mathtt{q}_1 := \mathtt{x}_2$, $\mathtt{b}_1$ indicates a potential use of the previous definition of $\mathtt{b}$ and $\mathtt{b}_2$ a potentially subsequent re-definition of $\mathtt{b}$. The opportunities for strong updates at a χ function are explored below.

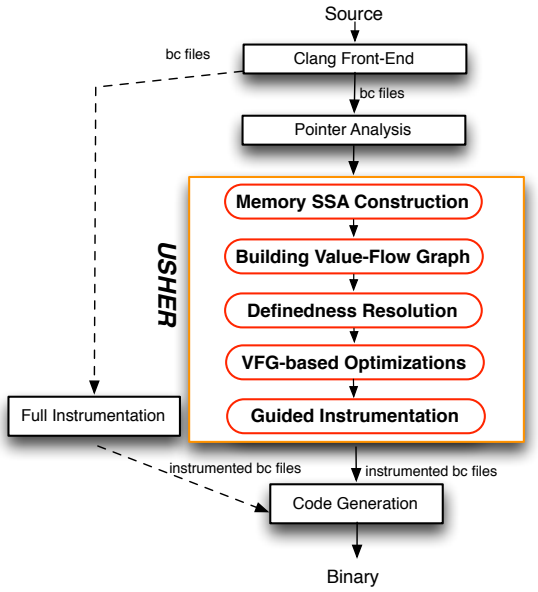


Figure 3: The USHER value-flow analysis framework.

$$
\begin{array}{rcll}
F & ::= & \ldots & \\
 & ::= & \text{def } f(a\ [\overline{\rho}])\{\ldots \text{ ret } r\ [\overline{\rho}];\} & \text{(virtual input and output parameters)} \\
stmt & ::= & \ldots & \\
 & | & x := \text{alloc}^{-}_{\rho}\ [\rho := \chi(\rho)] & \text{(allocation)} \\
 & | & x := *y\ \ \overline{[\mu(\rho)]} & \text{(load)} \\
 & | & *x := y\ \ \overline{[\rho := \chi(\rho)]} & \text{(store)} \\
 & | & x\ [\overline{\rho}] := f(y\ [\overline{\rho}]) & \text{(call)} \\
 & | & v := \phi(v, v) & \text{(phi)}
\end{array}
$$

$$v \in Var^{TL} \cup Var^{AT}$$

Figure 4: The TINYC language in SSA form.

```
...
a := alloc_b^F;
... := foo(a);
...
def foo(q) {
  x := *q;
  if x goto l;
  t := 10;
  x := x ⊗ t;
  *q := x;
l: ret x;
}
```

(a) TINYC

```
...
a1 := alloc_b^F  [b2 := χ(b1)];
... := foo(a1 [b2]);
...
def foo(q1 [b1]) {
  x1 := *q1 [μ(b1)];
  if x1 goto l';
  t1 := 10;
  x2 := x1 ⊗ t1;
  *q1 := x2  [b2 := χ(b1)];
l': x3 := φ(x1, x2);
  b3 := φ(b1, b2);
  ret x3 [b3];
}
```

(b) SSA

Figure 5: A TINYC program and its SSA form.

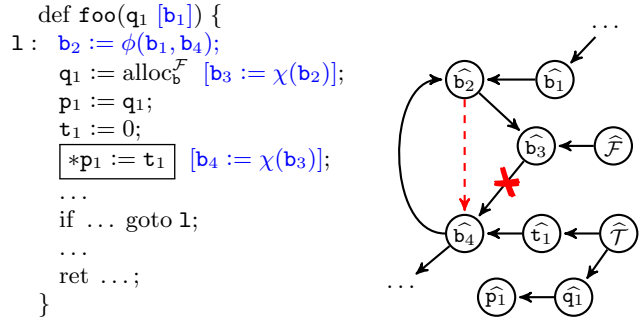
3.2 Building Value-Flow Graph

During this phase, USHER builds a value-flow graph for a program to capture the def-use chains both within a function and across the function boundaries in a program. What is novel about this phase is that two types of strong updates are considered for store statements.

For each definition v_r in the SSA form of a program, where r is the version of v, we write $\widehat{v_r}$ for its node in the VFG. We sometimes elide the version number when the context is clear. A *value-flow edge* $\widehat{v_m} \hookleftarrow \widehat{v_n}$ indicates a data dependence of v_m on v_n. Since we are only concerned with checking the definedness of a value used at a critical operation, it suffices to build the VFG only for the part of the program dependent by all critical operations.

For an allocation site $x_r := \text{alloc}^I_\rho\ [\rho_m := \chi(\rho_n)]$, where $I \in \{\mathcal{T}, \mathcal{F}\}$, we add $\widehat{x_r} \hookleftarrow \widehat{\mathcal{T}}$ (since x_r points to ρ), $\widehat{\rho_m} \hookleftarrow \widehat{I}$ and $\widehat{\rho_m} \hookleftarrow \widehat{\rho_n}$. Here, $\widehat{\mathcal{T}}$ and $\widehat{\mathcal{F}}$ are two special nodes, called the *root nodes* in the VFG, with $\widehat{\mathcal{T}}$ representing a defined value and $\widehat{\mathcal{F}}$ an undefined value.

For an assignment representing a copy, binary operation,



(a) TinyC (b) VFG

Figure 6: A semi-strong update performed at $*p_1 := t_1$. With a weak update, $\widehat{b_4} \hookleftarrow \widehat{b_3}$ would be introduced. With a semi-strong update, this edge is replaced (indicated by a cross) by $\widehat{b_4} \hookleftarrow \widehat{b_2}$ (indicated by the dashed arrow) so that $\widehat{b_3} \hookleftarrow \widehat{\mathcal{F}}$ is bypassed.

load or ϕ statement of the form $x_m := \ldots$, we add $\widehat{x_m} \hookleftarrow \widehat{y_n}$ for every use of y_n on the right-hand side of the assignment. Given $a_2 := b_3 \otimes c_4$, for example, $\widehat{a_2} \hookleftarrow \widehat{b_3}$ and $\widehat{a_2} \hookleftarrow \widehat{c_4}$ will be added. Given $d_4 := 10$, $\widehat{d_4} \hookleftarrow \widehat{\mathcal{T}}$ will be created.

For stores, we consider both traditional strong and weak updates as well as a new semi-strong update. Consider a store $*x_s = y_t\ [\rho_m := \chi(\rho_n)]$. If x_s uniquely points to a concrete location ρ, ρ_m can be *strongly updated*. In this case, ρ_m receives whatever y_t contains and the value flow from ρ_n is killed. So only $\widehat{\rho_m} \hookleftarrow \widehat{y_t}$ is added. Otherwise, ρ_m must incorporate the value flow from ρ_n, by also including $\widehat{\rho_m} \hookleftarrow \widehat{\rho_n}$. As a result, ρ_m can only be *weakly updated*.

Presently, Usher uses a pointer analysis that does not provide must-alias information. We improve precision by also performing a *semi-strong update* for a store $*x_s := y_t\ [\rho_m := \chi(\rho_n)]$, particularly when it resides in a loop. Suppose there is an allocation site $z_r := \text{alloc}_\rho^I\ [\_ := \chi(\rho_j)]$ such that $\widehat{z_r}$ dominates $\widehat{x_s}$ in the VFG, which implies that $z_r := \text{alloc}_\rho^I$ dominates $*x_s := y_t$ in the CFG (Control-Flow Graph) of the program as both z_r and x_s are top-level variables. This means that x_s uniquely points to ρ created at the allocation site. Instead of adding $\widehat{\rho_m} \hookleftarrow \widehat{y_t}$ and $\widehat{\rho_m} \hookleftarrow \widehat{\rho_n}$ by performing a weak update, we will add $\widehat{\rho_m} \hookleftarrow \widehat{y_t}$ and $\widehat{\rho_m} \hookleftarrow \widehat{\rho_j}$.

Consider an example given in Figure 6, where `foo` may be called multiple times so that the address-taken variable `b` is both used (read) and modified inside. At the store $*p_1 := t_1$, p_1 points to an abstract location. So a strong update is impossible. If a weak update is applied, $\widehat{b_4} \hookleftarrow \widehat{t_1}$ and $\widehat{b_4} \hookleftarrow \widehat{b_3}$ will be introduced, causing Usher to conclude that b_4 may be undefined due to the presence of $\widehat{b_3} \hookleftarrow \widehat{\mathcal{F}}$. Since $\widehat{q_1}$ dominates $\widehat{p_1}$, a semi-strong update can be performed at the store $*p_1 := t_1$. Instead of $\widehat{b_4} \hookleftarrow \widehat{b_3}$, which is introduced by a weak update, $\widehat{b_4} \hookleftarrow \widehat{b_2}$ is added, so that $\widehat{b_3} \hookleftarrow \widehat{\mathcal{F}}$ will be bypassed. Usher can then more precisely deduce that b_4 is defined as long as b_1 is.

Finally, we discuss how to add value-flow edges across the function boundaries. Consider a function definition $\text{def}\, f(a_1\, [\rho_1^1, \rho_1^2, \ldots]) \{\ldots\ \text{ret}\, r_s\, [\rho_{i_1}^1, \rho_{i_2}^2, \ldots];\}$, where ρ_1^k ($\rho_{i_k}^k$) is the k-th virtual input (output) parameter with version 1 (i_k). For each call site $x_t\, [\rho_{j_1}^1, \rho_{j_2}^2, \ldots] = f(y_m\, [\rho_{h_1}^1, \rho_{h_2}^2, \ldots])$, we add $\widehat{a_1} \hookleftarrow \widehat{y_m}$ and $\widehat{\rho_1^k} \hookleftarrow \widehat{\rho_{h_k}^k}$ (for every k) to connect each actual argument to its corresponding formal parameter. Similarly, we also propagate each output parameter to the call site where it is visible, by adding $\widehat{x_t} \hookleftarrow \widehat{r_s}$ and $\widehat{\rho_{j_k}^k} \hookleftarrow \widehat{\rho_{i_k}^k}$ (for every k).

3.3 Definedness Resolution

Presently, Usher instruments every function in a program only once (without cloning the function). Therefore, the definedness of all the variables (i.e., values) in the VFG of the program can be approximated by a graph reachability analysis, context-sensitively by matching call and return edges to rule out unrealizable interprocedural flows of values in the standard manner [18, 23, 25, 29, 33].

Let Γ be a function mapping the set of nodes in the VFG to $\{\bot, \top\}$. The definedness, i.e., *state* of a node $\widehat{v}$ is $\Gamma(\widehat{v}) = \bot$ if it is reachable by the root $\widehat{\mathcal{F}}$ and $\Gamma(\widehat{v}) = \top$ otherwise (i.e., if it is reachable only by the other root, $\widehat{\mathcal{T}}$).

3.4 Guided Instrumentation

Instead of shadowing all variables and statements in a program, Usher solves a graph reachability problem on its VFG by identifying only a subset of these to be instrumented at run time. The instrumentation code generated by Usher is sound as long as the underlying pointer analysis used is. This ensures that all possible undefined values flowing into every critical operation in a program are tracked at run time.

During this fourth phase (and also the last phase in Section 3.5), Usher works on a program in SSA form. To avoid cluttering, we often refer to an SSA variable with its version being elided since it is deducible from the context.

A statement may need to be shadowed only if the value $\widehat{v}$ defined (directly/indirectly) by the statement can reach a node $\widehat{x}$ that satisfies $\Gamma(\widehat{x}) = \bot$ in the VFG such that x is used in a critical statement. A sound instrumentation implies that all shadow values accessed by any shadow statement at run time are well-defined.

Given a statement $\ell : s$, we formally define an *instrumentation item* for $\ell : s$ as a pair $\langle \overleftarrow{\ell}, \underline{s} \rangle$ or $\langle \overrightarrow{\ell}, \underline{s} \rangle$, indicating that the shadow operation (or statement) $\underline{s}$ for s is inserted just before or after ℓ (with s omitted). The instrumentation item sets for different types of statements are computed according to the instrumentation rules given in Figure 7.

The deduction rules are formulated in terms of

$$\mathcal{P}, \Gamma \quad \vdash \quad \widehat{v} \Downarrow \Sigma_{\widehat{v}} \tag{1}$$

where $\Sigma_{\widehat{v}}$ is the set of instrumentation items that enables the flows of undefined values into node $\widehat{v}$ to be tracked soundly via shadow propagations. This is achieved by propagating the Σ's of $\widehat{v}$'s predecessors in the VFG into $\widehat{v}$ and also adding relevant new instrumentation items for $\widehat{v}$. Here, $\mathcal{P}$ is a given program in SSA form. In addition, $\mathcal{P}(code)$ holds if the block of statements, denoted *code*, exists in $\mathcal{P}$.

In shadow-memory-based instrumentation, a runtime shadow map, denoted σ, is maintained for mapping variables (or precisely their locations) to (the locations of) their shadow variables. In addition, $\mathcal{E}$ records at run time whether a critical statement has accessed an undefined value or not.

The guided instrumentation for $\mathcal{P}$ is precisely specified as the union of Σ's computed by applying the rules in Figure 7 to all nodes representing the uses at critical operations. In

$\boxed{\mathcal{P}, \Gamma \vdash \widehat{v} \Downarrow \Sigma_{\widehat{v}}}$

$$[\top\text{-Check}]\ \frac{s \in \{\_ := *x,\ *x := \_,\ \text{if } x \text{ goto } \_\} \qquad \mathcal{P}(\ell : s) \qquad \Gamma(\widehat{x}) = \top}{\mathcal{P}, \Gamma \vdash \widehat{\ell^x} \Downarrow \emptyset}$$

$$[\top\text{-Assign}]\ \frac{s \in \{x := n/y,\ x := \text{alloc}^{\_},\ x := \_ \otimes \_,\ x := *\_,\ x\ [\_] := f(\_)\} \qquad \mathcal{P}(\ell : s) \qquad \Gamma(\widehat{x}) = \top}{\mathcal{P}, \Gamma \vdash \widehat{x} \Downarrow \{\langle \overrightarrow{\ell},\ \sigma(x) := \mathcal{T}\rangle\}}$$

$$[\top\text{-Para}]\ \frac{\mathcal{P}(\text{def } f(a\ [\_])\{\ell : \_;\ \ldots\}) \qquad \Gamma(\widehat{a}) = \top}{\mathcal{P}, \Gamma \vdash \widehat{a} \Downarrow \{\langle \overleftarrow{\ell},\ \sigma(a) := \mathcal{T}\rangle\}}$$

$$[\top\text{-Alloc}]\ \frac{\mathcal{P}(\ell : x := \text{alloc}^{\mathcal{T}}_{\rho}\ [\rho_m := \chi(\_)]) \qquad \Gamma(\widehat{\rho_m}) = \top}{\mathcal{P}, \Gamma \vdash \widehat{\rho_m} \Downarrow \{\langle \overrightarrow{\ell},\ \sigma(*x) := \mathcal{T}\rangle\}}$$

$$[\top\text{-Store}^{SU}]\ \frac{\mathcal{P}(\ell : *x := \_\ [\rho_m := \chi(\_)]) \qquad \Gamma(\widehat{\rho_m}) = \top \qquad \widehat{\rho_m} \not\leftrightarrow \widehat{\rho_{\_}}}{\mathcal{P}, \Gamma \vdash \widehat{\rho_m} \Downarrow \{\langle \overrightarrow{\ell},\ \sigma(*x) := \mathcal{T}\rangle\}}$$

$$[\top\text{-Store}^{WU/SemiSU}]\ \frac{\mathcal{P}(\_ : *\_ := \_\ [\_, \rho_m := \chi(\_), \_]) \qquad \Gamma(\widehat{\rho_m}) = \top \qquad \widehat{\rho_m} \leftrightarrow \widehat{\rho_n} \qquad \mathcal{P}, \Gamma \vdash \widehat{\rho_n} \Downarrow \Sigma_{\widehat{\rho_n}}}{\mathcal{P}, \Gamma \vdash \widehat{\rho_m} \Downarrow \Sigma_{\widehat{\rho_n}}}$$

$$[\bot\text{-Check}]\ \frac{s \in \{\_ := *x,\ *x := \_,\ \text{if } x \text{ goto } \_\} \qquad \mathcal{P}(\ell : s) \qquad \Gamma(\widehat{x}) = \bot \qquad \mathcal{P}, \Gamma \vdash \widehat{x} \Downarrow \Sigma_{\widehat{x}}}{\mathcal{P}, \Gamma \vdash \widehat{\ell^x} \Downarrow \Sigma_{\widehat{x}} \cup \{\langle \overleftarrow{\ell},\ \mathcal{E}(\ell) := (\sigma(x) = \mathcal{F})\rangle\}}$$

$$[\bot\text{-VCopy}]\ \frac{\mathcal{P}(\ell : x := y) \qquad \Gamma(\widehat{x}) = \bot \qquad \mathcal{P}, \Gamma \vdash \widehat{y} \Downarrow \Sigma_{\widehat{y}}}{\mathcal{P}, \Gamma \vdash \widehat{x} \Downarrow \Sigma_{\widehat{y}} \cup \{\langle \overrightarrow{\ell},\ \sigma(x) := \sigma(y)\rangle\}}$$

$$[\bot\text{-Bop}]\ \frac{\mathcal{P}(\ell : x := y \otimes z) \qquad \Gamma(\widehat{x}) = \bot \qquad \mathcal{P}, \Gamma \vdash \widehat{y} \Downarrow \Sigma_{\widehat{y}} \qquad \mathcal{P}, \Gamma \vdash \widehat{z} \Downarrow \Sigma_{\widehat{z}}}{\mathcal{P}, \Gamma \vdash \widehat{x} \Downarrow \Sigma_{\widehat{y}} \cup \Sigma_{\widehat{z}} \cup \{\langle \overrightarrow{\ell},\ \sigma(x) := \sigma(y) \wedge \sigma(z)\rangle\}}$$

$$[\bot\text{-Alloc}]\ \frac{\mathcal{P}(\ell : x := \boxed{\text{alloc}^{\mathcal{T}}_{\rho}/\text{alloc}^{\mathcal{F}}_{\rho}}\ [\rho_m := \chi(\rho_n)]) \qquad \Gamma(\widehat{\rho_m}) = \bot \qquad \mathcal{P}, \Gamma \vdash \widehat{\rho_n} \Downarrow \Sigma_{\widehat{\rho_n}}}{\mathcal{P}, \Gamma \vdash \widehat{\rho_m} \Downarrow \Sigma_{\widehat{\rho_n}} \cup \{\langle \overrightarrow{\ell},\ \sigma(*x) := \boxed{\mathcal{T}/\mathcal{F}}\ \rangle\}}$$

$$[\bot\text{-Para}]\ \frac{\mathcal{P}(\text{def } f(a\ [\_])\{\ell : \_; \ldots\}) \qquad \Gamma(\widehat{a}) = \bot \qquad \forall\, \ell_i \in \mathcal{C}_f,\ \mathcal{P}(\ell_i : \_ := f(y^i\ [\_])) :\ \mathcal{P}, \Gamma \vdash \widehat{y^i} \Downarrow \Sigma_{\widehat{y^i}}}{\mathcal{P}, \Gamma \vdash \widehat{a} \Downarrow (\bigcup_i \Sigma_{\widehat{y^i}}) \cup \{\langle \overleftarrow{\ell},\ \sigma(a) := \sigma_g\rangle, \langle \overleftarrow{\ell_i},\ \sigma_g := \sigma(y^i)\rangle\}}$$

$$[\bot\text{-Ret}]\ \frac{\mathcal{P}(\ell : x\ [\_] := f(\_)) \qquad \mathcal{P}(\text{def } f(\_)\{\ldots\ \ell' : \_; \text{ret } r\ [\_];\}) \qquad \Gamma(\widehat{x}) = \bot \qquad \mathcal{P}, \Gamma \vdash \widehat{r} \Downarrow \Sigma_{\widehat{r}}}{\mathcal{P}, \Gamma \vdash \widehat{x} \Downarrow \Sigma_{\widehat{r}} \cup \{\langle \overrightarrow{\ell},\ \sigma(x) := \sigma_g\rangle, \langle \overrightarrow{\ell'},\ \sigma_g := \sigma(r)\rangle\}}$$

$$[\bot\text{-Load}]\ \frac{\mathcal{P}(\ell : x := *y\ [\mu(\rho^1_{\_}), \mu(\rho^2_{\_}), \ldots]) \qquad \forall \rho^i_{\_} :\ \mathcal{P}, \Gamma \vdash \widehat{\rho^i_{\_}} \Downarrow \Sigma_{\widehat{\rho^i_{\_}}} \qquad \Gamma(\widehat{x}) = \bot}{\mathcal{P}, \Gamma \vdash \widehat{x} \Downarrow (\bigcup_i \Sigma_{\widehat{\rho^i_{\_}}}) \cup \{\langle \overrightarrow{\ell},\ \sigma(x) := \sigma(*y)\rangle\}}$$

$$[\bot\text{-Store}^{SU/\boxed{WU/SemiSU}}]\ \frac{\mathcal{P}(\ell : *x := y\ [\_, \rho_m := \chi(\_), \_]) \qquad \Gamma(\widehat{\rho_m}) = \bot \quad \mathcal{P}, \Gamma \vdash \widehat{y} \Downarrow \Sigma_{\widehat{y}} \quad \boxed{\widehat{\rho_m} \leftrightarrow \widehat{\rho_n} \quad \mathcal{P}, \Gamma \vdash \widehat{\rho_n} \Downarrow \Sigma_{\widehat{\rho_n}}}}{\mathcal{P}, \Gamma \vdash \widehat{\rho_m} \Downarrow \Sigma_{\widehat{y}} \cup \{\langle \overrightarrow{\ell},\ \sigma(*x) := \sigma(y)\rangle\}\ \boxed{\cup \Sigma_{\widehat{\rho_n}}}}$$

$$[\text{Phi}]\ \frac{\mathcal{P}(\_ : v_l := \phi(v_m, v_n)) \qquad \mathcal{P}, \Gamma \vdash \widehat{v_m} \Downarrow \Sigma_{\widehat{v_m}} \qquad \mathcal{P}, \Gamma \vdash \widehat{v_n} \Downarrow \Sigma_{\widehat{v_n}}}{\mathcal{P}, \Gamma \vdash \widehat{v_l} \Downarrow \Sigma_{\widehat{v_m}} \cup \Sigma_{\widehat{v_n}}}$$

$$[\text{VPara}]\ \frac{\mathcal{P}(\text{def } f(\_\ [\_, \rho_m, \_])\ \{\ldots\}) \qquad \forall\, \widehat{\rho_m} \leftrightarrow \widehat{\rho_i} :\ \mathcal{P}, \Gamma \vdash \widehat{\rho_i} \Downarrow \Sigma_{\widehat{\rho_i}}}{\mathcal{P}, \Gamma \vdash \widehat{\rho_m} \Downarrow \bigcup_i \Sigma_{\widehat{\rho_i}}}$$

$$[\text{VRet}]\ \frac{\mathcal{P}(\_ : \_\ [\_, \rho_m, \_] := f(\_)) \qquad \widehat{\rho_m} \leftrightarrow \widehat{\rho_n} \qquad \mathcal{P}, \Gamma \vdash \widehat{\rho_n} \Downarrow \Sigma_{\widehat{\rho_n}}}{\mathcal{P}, \Gamma \vdash \widehat{\rho_m} \Downarrow \Sigma_{\widehat{\rho_n}}}$$

Auxiliaries: σ_g is a global variable introduced at run time to shadow parameter passing.
$\mathcal{C}_f := \{\ell_i \mid \ell_i \text{ is a call site for function } f\}$.

Figure 7: Instrumentation rules.

[⊤-Check] and [⊥-Check], $\widehat{\ell^x}$ denotes a virtual node (due to the existence of a virtual assignment of the form $\ell^x := x$) associated with the critical statement ℓ to ease the presentation.

Different propagation schemes are used for ⊤-nodes $\widehat{v}$ (where $\Gamma(\widehat{v}) = \top$) and ⊥-nodes $\widehat{v}$ (where $\Gamma(\widehat{v}) = \bot$). The rules are divided into three sections (separated by the dashed lines): (1) those prefixed by ⊤ for ⊤-nodes, (2) those prefixed by ⊥ for ⊥-nodes, and (3) the rest for some "virtual" nodes introduced for handling control-flow splits and joins.

Special attention should be paid to the rules (that apply to ⊤-nodes only), where a shadow location can be strongly updated. The remaining rules are straightforward. Consider a statement where $\sigma(v)$ needs to be computed for a variable v at run time. We say that $\sigma(v)$ can be *strongly updated* if $\sigma(v) := \mathcal{T}$ can be set directly at run time to indicate that v is defined at that point so that the (direct or indirect) predecessors of $\widehat{v}$ in the VFG do not have to be instrumented with respect to v at this particular statement.

⊤-Nodes. Let us first consider the rules for ⊤-nodes. The value flow of a (top-level or address-taken) variable v is mimicked exactly by that of its shadow $\sigma(v)$. There are two cases in which a strong update to $\sigma(v)$ can be safely performed. For top-level variables, this happens in [⊤-Assign] and [⊤-Para]), which are straightforward to understand.

For address-taken variables, strong updates are performed in [⊤-Alloc]) and [⊤-Store$^{\text{SU}}$] but not in [⊤-Store$^{\text{WU/SemiSU}}$]. For an allocation site $x := \text{alloc}_\rho^{\mathcal{T}}$ $[\rho_m := \chi(\_)]$, such that $\Gamma(\widehat{\rho_m}) = \top$, $*x$ uniquely represents the location ρ_m, which contains a well-defined value. Therefore, $\sigma(*x)$ can be strongly updated, by setting $\sigma(*x) := \mathcal{T}$ ([⊤-Alloc]).

Let us consider an indirect def ρ_m at a store, where $\widehat{\rho_m}$ is a ⊤-node. As discussed in Section 3.2, $\widehat{\rho_m}$ has at most two predecessors. One predecessor represents the variable, say y_t, on the right-hand side of the store. The shadow propagation for y_t is not needed since $\Gamma(\widehat{\rho_m}) = \top$ implies $\Gamma(\widehat{y_t}) = \top$. The other predecessor represents an older version of ρ, denoted ρ_n. If $\widehat{\rho_m} \hookleftarrow \widehat{\rho_n}$ is absent, then [⊤-Store$^{\text{SU}}$] applies. Otherwise, [⊤-Store$^{\text{WU/SemiSU}}$] applies. In the former case, $\sigma(*x) := \mathcal{T}$ is strongly updated as x uniquely points to a concrete location ρ. However, the same cannot happen in [⊤-Store$^{\text{WU/SemiSU}}$] since the resulting instrumentation would be incorrect otherwise. Consider the following code snippet:

$$*\mathtt{p}_2 := \mathtt{t}_1 \;\; [\mathtt{b}_3 := \chi(\mathtt{b}_2),\; \mathtt{c}_4 := \chi(\mathtt{c}_3)];$$
$$\ldots := *\mathtt{q}_3 \;\; [\mu(\mathtt{b}_3];$$

Even $\Gamma(\widehat{\mathtt{b}_3}) = \Gamma(\widehat{\mathtt{c}_4}) = \top$, we cannot directly set $\sigma(*\mathtt{p}) := \mathcal{T}$ due to the absence of strong updates to `b` and `c` at the store. During a particular execution, it is possible that $\mathtt{p}_2$ points to `c` but $\mathtt{q}_3$ points to `b`. In this case, $*\mathtt{p}_2$ is not a definition for `b`. If `b` needs to be shadowed at the load, its shadow $\sigma(\mathtt{b})$ must be properly initialized earlier and propagated across the store to ensure its well-definedness at the load.

Finally, a runtime check is not needed at a critical operation when a defined value is used ([⊤-Check]).

⊥-Nodes. Now let us discuss the rules for ⊥-nodes. The instrumentation code is generated as in full instrumentation, requiring the instrumentation items for its predecessors to be generated to enable shadow propagations into this node. [⊥-VCopy] and [⊥-Bop]) are straightforward to understand. For an allocation site $x := \text{alloc}_\rho^{\mathcal{T}}(\text{alloc}_\rho^{\mathcal{F}})$ $[\rho_m := \chi(\rho_n)]$, such that $\Gamma(\widehat{\rho_m}) = \bot$, $\sigma(*x)$, i.e., the shadow for the object currently allocated at the site, is strongly updated to be $\mathcal{T}$ ($\mathcal{F}$). In addition, the older version ρ_n is tracked as well.

The standard parameter passing for a function is instrumented so that the value of the shadow of its actual argument at every call site is propagated into the shadow of the (corresponding) formal parameter ([⊥-Para]). This is achieved by using an auxiliary global variable σ_g to relay an shadow value across two different scopes. Retrieving a value returned from a function is handled similarly ([⊥-Ret]).

At a load $x := *y$, where $\Gamma(\widehat{x}) = \bot$, all the indirect uses made via $*y$ must be tracked separately to enable the shadow propagation $\sigma(x) := \sigma(*y)$ for the load ([⊥-Load]).

In [⊥-Store$^{\text{SU/}\boxed{\text{WU/SemiSU}}}$], strong updates to shadow locations cannot be safely performed. In particular, the value flow from the right-hand side y of a store must also be tracked, unlike in [⊤-Store$^{\text{SU}}$] and [⊤-Store$^{\text{WU/SemiSU}}$].

When an undefined value x may be potentially used at a critical statement at ℓ, a runtime check must be performed at the statement ([⊥-Check]). In this case, $\mathcal{E}(\ell)$ is set to true if and only if $\sigma(x)$ evaluates to $\mathcal{F}$.

Virtual Nodes. For the "virtual" value-flow edges added due to ϕ and parameter passing for virtual input and output parameters, the instrumentation items required will be simply collected across the edges, captured by [Phi], [VPara] and [VRet]. During program execution, the corresponding shadow values will "flow" across such value-flow edges.

3.5 VFG-based Optimizations

Our VFG representation is general as it allows various instrumentation-reducing optimizations to be developed. Below we describe two optimizations, developed based on the concept of *Must Flow-from Closure* (MFC), denoted ∇.

DEFINITION 2 (MFC). $\nabla_{\widehat{x}}$ for a top-level variable x is:

$$\nabla_{\widehat{x}} := \begin{cases} \{\widehat{x}\} \cup \nabla_{\widehat{y}} \cup \nabla_{\widehat{z}}, & \mathcal{P}(x := y \otimes z) \\ \{\widehat{x}\} \cup \nabla_{\widehat{y}}, & \mathcal{P}(x := y) \\ \{\widehat{x}, \widehat{\mathcal{T}}\}, & \mathcal{P}(x := n) \text{ or } \mathcal{P}(x := \text{alloc}_{\_}) \\ \{\widehat{x}\}, & \text{otherwise} \end{cases}$$

It is easy to see that $\nabla_{\widehat{x}}$ is a DAG (directed acyclic graph), with $\widehat{x}$ as the (sole) sink and one or more sources (i.e., the nodes without incoming edges). In addition, $\Gamma(\widehat{x}) = \top$ if and only if $\Gamma(\widehat{y}) = \top$ for all nodes $\widehat{y}$ in $\nabla_{\widehat{x}}$.

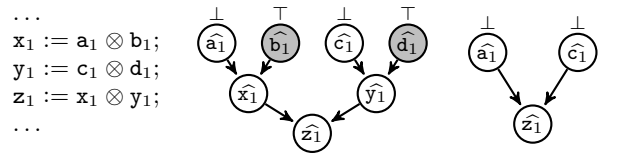
$\nabla_{\widehat{x}}$ contains only top-level variables because loads and stores cannot be bypassed during shadow propagations.

3.5.1 Optimization I: Value-Flow Simplification

This optimization (referred to as *Opt I* later) aims to reduce shadow propagations in an MFC. For each $\nabla_{\widehat{x}}$, the shadow value $\sigma(x)$ of a top-level variable x is a conjunct of the shadow values of its source nodes. Thus, it suffices to propagate directly the shadow values of the sources s, such that $\Gamma(\widehat{s}) = \bot$, to $\widehat{x}$, as illustrated in Figure 8.

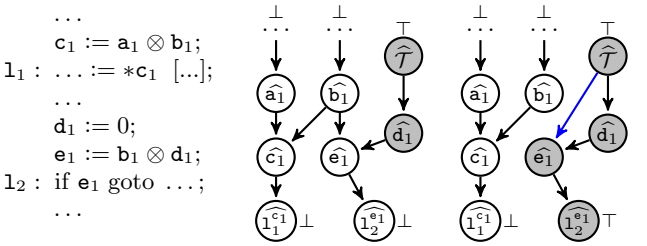
3.5.2 Optimization II: Redundant Check Elimination

Our second optimization (*Opt II*) is more elaborate but also conceptually simple. The key motivation is to reduce instrumentation overhead by avoiding spurious error messages. If an undefined value can be detected at a critical statement, then its rippling effects on the other parts of the program (e.g., other critical statements) can be suppressed.



(a) TinyC (b) $\nabla_{\widehat{z_1}}$ (c) Simplified $\nabla_{\widehat{z_1}}$

Figure 8: An example of value-flow simplification.



(a) TinyC (b) VFG (c) Modified VFG

Figure 9: An example for illustrating redundant check elimination, where $\mathtt{l}_1$ is assumed to dominate $\mathtt{l}_2$ in the CFG of the program. If $\mathtt{b}_1$ has an undefined value, then the error can be detected at both $\mathtt{l}_1$ and $\mathtt{l}_2$. The check at $\mathtt{l}_2$ can therefore be disabled by a simple modification of the original VFG.

The basic idea is illustrated in Figure 9. There are two runtime checks at $\mathtt{l}_1$ and $\mathtt{l}_2$, where $\mathtt{l}_1$ is known to dominate $\mathtt{l}_2$ in the CFG for the code in Figure 9(a). According to its VFG in Figure 9(b), $\mathtt{b}_1$ potentially flows into both $\mathtt{c}_1$ and $\mathtt{e}_1$. If $\mathtt{b}_1$ is the culprit for the use of an undefined value via $\mathtt{c}_1$ at $\mathtt{l}_1$, $\mathtt{b}_1$ will also cause an uninitialized read via $\mathtt{e}_1$ at $\mathtt{l}_2$. If we perform definedness resolution on the VFG in Figure 9(c) modified from Figure 9(b), by replacing $\widehat{\mathtt{e}_1} \hookleftarrow \widehat{\mathtt{b}_1}$ with $\widehat{\mathtt{e}_1} \hookleftarrow \widehat{\top}$, then no runtime check at $\mathtt{l}_2$ is necessary (since [$\top$-Check] is applicable to $\mathtt{l}_2$ when $\Gamma(\mathtt{e}_1) = \top$).

As shown in Algorithm 1, we perform this optimization by modifying the VFG of a program and then recomputing Γ. If an undefined value *definitely* flows into a critical statement s via either a top-level variable in $\nabla_{\widehat{x}}$ or possibly an address-taken variable ρ_m (lines 3 – 4), then the flow of this undefined value into another node $\widehat{r}$ outside $\nabla_{\widehat{x}}$ (lines 5 – 6) such that s dominates s_r, where $\widehat{r}$ is defined, can be redirected from $\widehat{\top}$ (lines 7 – 8). As some value flows from address-taken variables may have been cut (line 9), Usher must perform its guided instrumentation on the VFG (obtained without this optimization) by using Γ obtained here to ensure that all shadow values are correctly initialized.

Algorithm 1 Redundant Check Elimination

begin

1. $G \leftarrow$ the VFG of the program $\mathcal{P}$;
2. **foreach** *top-level variable* $x \in Var^{TL}$ *used at a critical statement, denoted* s*, in* $\mathcal{P}$ **do**
3. $\quad \nabla_{\widehat{x}} \leftarrow$ MFC computed for $\widehat{x}$ in G;
4. $\quad \nabla'_{\widehat{x}} \leftarrow \nabla_{\widehat{x}} \cup \{\widehat{\rho_m} \mid \widehat{y} \in \nabla_{\widehat{x}},\ \mathcal{P}(y := *z\ [\mu(\rho_m)]),\ \rho_m \in Var^{AT}$ represents a concrete location$\}$;
5. $\quad \mathcal{R}_{\widehat{x}} \leftarrow \{\widehat{r} \mid \widehat{t} \in \nabla'_{\widehat{x}},\ \widehat{r} \notin \nabla'_{\widehat{x}},\ \widehat{r} \hookleftarrow \widehat{t}$ in $G\}$;
6. $\quad$ **foreach** *statement* s_r*, where* $\widehat{r} \in \mathcal{R}_{\widehat{x}}$ *is defined* **do**
7. $\quad\quad$ **if** s *dominates* s_r *in the CFG of* $\mathcal{P}$ **then**
8. $\quad\quad\quad$ Replace every $\widehat{r} \hookleftarrow \widehat{t}$, where $\widehat{t} \in \nabla'_{\widehat{x}}$, by $\widehat{r} \hookleftarrow \widehat{\top}$ in G;
9. Perform definedness resolution to obtain Γ on G;

4. EVALUATION

The main objective is to demonstrate that by performing a value-flow analysis, Usher can significantly reduce instrumentation overhead of MSan, a state-of-the-art source-level instrumentation tool for detecting uses of undefined values.

4.1 Implementation

We have implemented Usher in LLVM (version 3.3), where MSan is released. Usher uses MSan's masked offset-based shadow memory scheme for instrumentation and its runtime library to summarize the side effects of external functions on the shadow memory used.

Usher performs an interprocedural whole-program analysis to reduce instrumentation costs. All source files of a program are compiled and then merged into one bitcode file (using LLVM-link). The merged bitcode is transformed by iteratively inlining the functions with at least one function pointer argument to simplify the call graph (excluding those functions that are directly recursive). Then LLVM's mem2reg is applied to promote memory into (virtual) registers, i.e., generate SSA for top-level local variables. We refer to this optimization setting as O0+IM (i.e., LLVM's O0 followed by *I*nlining and *M*em2reg). Finally, LLVM's LTO (Link-Time Optimization) is applied.

For the pointer analysis phase shown in Figure 3, we have used an offset-based field-sensitive Andersen's pointer analysis [10]. Arrays are treated as a whole. 1-callsite-sensitive heap cloning is applied to allocation wrapper functions. 1-callsite context-sensitivity is configured for definedness resolution (Section 3.3). In addition, access-equivalent VFG nodes are merged by using the technique from [11].

In LLVM, all the global variables are accessed indirectly (via loads and stores) and are thus dealt with exactly as address-taken variables. Their value flows across the function boundaries are realized as virtual parameters as described in Figure 4 and captured by [VPara] and [VRet].

Like MSan, Usher's dynamic detection is bit-level precise [24], for three reasons. First, Usher's static analysis is conservative for bit-exactness. Second, at run time, every bit is shadowed and the shadow computations for bit operations in [$\perp$-Bop] (defined in Figure 7) are implemented as described in [24]. Finally, $\nabla_{\widehat{x}}$ given in Definition 2 is modified so that $\mathcal{P}(x := y \otimes z)$ holds when $\otimes$ is *not* a bitwise operation.

4.2 Platform and Benchmarks

All experiments are done on a machine equipped with a 3.00GHz quad-core Intel Core2 Extreme X9650 CPU and 8GB DDR2 RAM, running a 64-bit Ubuntu 10.10. All the 15 C benchmarks from SPEC CPU2000 are used and executed under their *reference* inputs. Some of their salient properties are given in Table 1 and explained below.

4.3 Methodology

Like MSan, Usher is designed to facilitate detection of

Benchmark	Size	Time	Mem	Var^{TL}	Var^{AT}					Stores			VFG			
	(KLOC)	(secs)	(MB)	(10^3)	Stack	Heap	Global	$\%\mathcal{F}$	$\overline{S}$	Total	%SU	%WU*	Nodes (10^3)	$\%\lvert\mathcal{B}\rvert$	S_∇(10^3)	$\lvert\mathcal{R}\rvert$ (10^3)
164.gzip	8.6	0.32	294	7	27	10	428	8	-	617	62	34	16	20	0.6	1.0
175.vpr	17.8	0.54	306	22	177	207	770	31	1.2	1,044	34	53	51	27	3.2	4.9
176.gcc	230.4	58.35	2,758	324	1,600	874	6,824	27	4.7	10,851	40	31	17,932	56	96.0	54.3
177.mesa	61.3	1.88	366	113	738	2,417	2,534	32	0.2	7,798	6	63	151	22	8.7	15.8
179.art	1.2	0.28	291	2	8	48	83	40	-	140	41	59	5	21	0.2	0.6
181.mcf	2.5	0.28	292	2	8	89	71	39	-	221	25	70	4	4	0.0	0.7
183.equake	1.5	0.29	293	4	32	29	122	33	-	189	26	68	6	11	0.6	0.9
186.crafty	21.2	0.70	315	29	71	528	1,460	29	-	2,215	63	28	103	34	2.1	4.5
188.ammp	13.4	0.57	307	26	76	342	416	50	4.9	1,291	11	76	55	32	4.7	6.7
197.parser	11.4	0.79	315	16	184	447	1,005	39	2.9	892	34	60	162	81	1.9	3.3
253.perlbmk	87.1	53.93	1,405	116	736	814	3,705	29	5.7	8,904	52	11	8,378	84	41.9	23.2
254.gap	71.5	19.21	701	125	54	4,101	4,313	49	-	4,378	16	28	1,941	48	49.5	21.8
255.vortex	67.3	11.15	601	76	3,576	1,548	3,602	45	-	6,169	70	5	2,483	78	7.7	11.6
256.bzip2	4.7	0.30	293	5	21	13	166	17	-	303	32	68	11	16	0.3	1.2
300.twolf	20.5	1.26	331	52	116	700	841	49	2.8	2,989	34	38	122	37	11.2	12.4
average	41.4	9.99	591	61	495	811	1,756	34	3.2	3,200	36	46	2,095	38	15.3	10.9

Table 1: Benchmark statistics under O0+IM. "$\%\mathcal{F}$" is the percentage of address-taken variables uninitialized when allocated. "$\overline{S}$" is the number of times our semi-strong update rule is applied per non-array heap allocation site. "%SU" is the percentage of stores with strong updates. "%WU*" is the percentage of stores $*x = y$ with x pointing to one address-taken variable (where weak updates would be performed if semi-strong updates are not applied). "$\%|\mathcal{B}|$" is the percentage of the VFG nodes reaching at least one critical statement, where a runtime check is needed. "S_∇" stands for the number of ∇'s simplified by Opt I. "$|\mathcal{R}|$" is the size of the union of $\mathcal{R}_{\widehat{x}}$'s for all $\widehat{x}$ defined in line 5 of Algorithm 1 by Opt II.

uninitialized variables. O0+IM represents an excellent setting for obtaining meaningful stack traces in error messages. In addition, LLVM under "-O1" or higher flags behaves non-deterministically on undefined (i.e., **undef**) values [35], making their runtime detection nondeterministic. Thus, we will focus on comparing MSan and Usher under O0+IM in terms of instrumentation overhead when both are implemented identically in LLVM except that their degrees of instrumentation differ. We will examine both briefly in Section 4.6 when higher optimization flags are used.

In addition, we will also highlight the importance of statically analyzing the value flows for address-taken variables and evaluate the benefits of our VFG-based optimizations.

4.4 Value-Flow Analysis

Table 1 presents some statistics for Usher's value-flow analysis under O0+IM. Usher is reasonably lightweight, consuming under 10 seconds (inclusive pointer analysis time) and 600 MB memory on average. The two worst performers are `176.gcc` and `253.perlbmk`, both taking nearly 1 minute and consuming ≈2.7 and ≈1.4 GB memory, respectively. The latter is more costly when compared to other benchmarks with similar sizes, since its larger VFG contains more interprocedural value-flow edges for its global and heap variables, which are both in Var^{AT}.

In Columns 5 – 8, some statistics for both Var^{TL} (containing the virtual registers produced by mem2reg) and Var^{AT} are given for each benchmark. In LLVM, global variables belong to Var^{AT} and are accessed via loads and stores. This explains why all benchmarks except `255.vortex` have more global variables than stack variables (that are not converted to virtual registers by mem2reg). However, at an allocation site $x := \text{alloc}^{-}_{\rho}$, where ρ is a global variable, x is a `const` top-level pointer and is thus always initialized ([⊤-Alloc]). So it needs not to be checked when used at a critical statement. In Column 9 (under "$\%\mathcal{F}$"), we see that 34% of the address-taken variables are not initialized when allocated on average. Note that heap objects allocated at a `calloc()` site or its wrappers are always initialized ([⊤-Alloc]).

In Columns 11 – 13, we can see some good opportunities for traditional strong updates, which kill undefined values to enable more ⊤-nodes to be discovered statically. According to the pointer analysis used [10], at 82% of the stores (on average), a (top-level) variable in Var^{TL} points to one single abstract object in Var^{AT}, with 82% being split into 36%, where strong updates are performed, and 46%, where weak updates would have to be applied. In Column 10, we see that the average number of times that our semi-strong update rule (introduced in Section 3.2) is applied, i.e., the average number of cuts made on the VFGs (highlighted by a cross in Figure 6) per non-array heap allocation site is 3.2.

By performing static analysis, Usher can avoid shadowing the statements that never produce any values consumed at a critical statement, where a runtime check is needed. Among all the VFG nodes (Column 14), only an average of 38% may need to be tracked (Column 15). In the second last column, the average number of simplified MFCs (Definition 2) by Opt I is 15251. In the last column, the average number of VFG nodes connected to $\widehat{\mathcal{T}}$ by Opt II, as illustrated in Figure 9, is 10859.

4.5 Instrumentation Overhead

Figure 10 compares Usher and MSan in terms of their relative slowdowns to the native (instrumentation-free) code for the 15 C benchmarks tested. MSan has an average slowdown of 302%, reaching 493% for `253.perlbmk`. With guided instrumentation, Usher has reduced MSan's average slowdown to 123%, with 340% for `253.perlbmk`. In addition, we have also evaluated three variations of Usher: (1) Usher^{TL}, which analyzes top-level variables only without performing Opt I and Opt II, which are described in Section 3.5, (2) Usher^{TL+AT}, which is Usher^{TL} extended

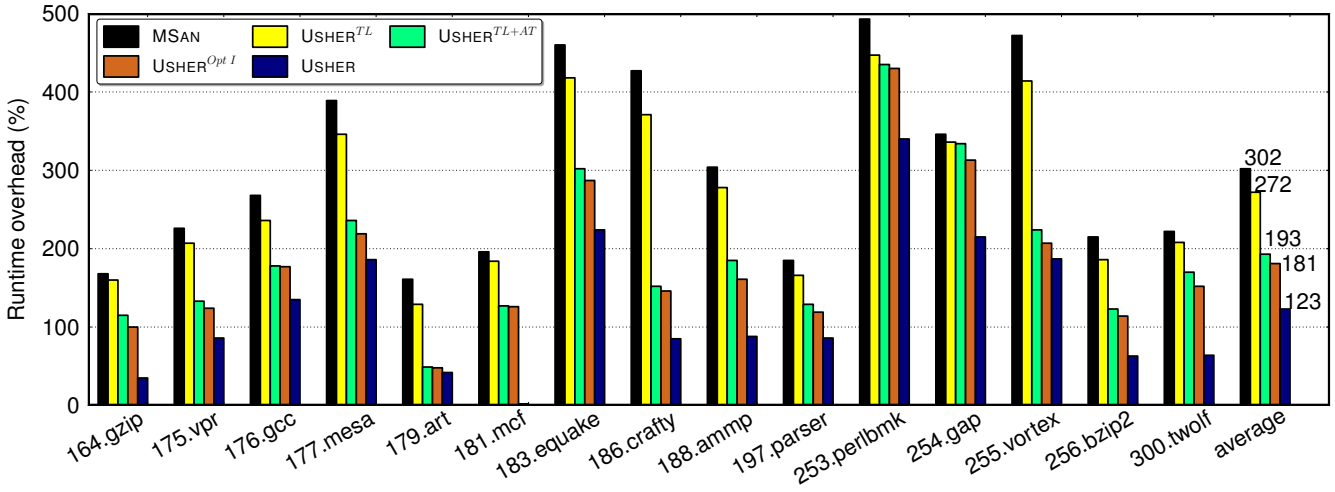


Figure 10: Execution time slowdowns (normalized with respect to native code).

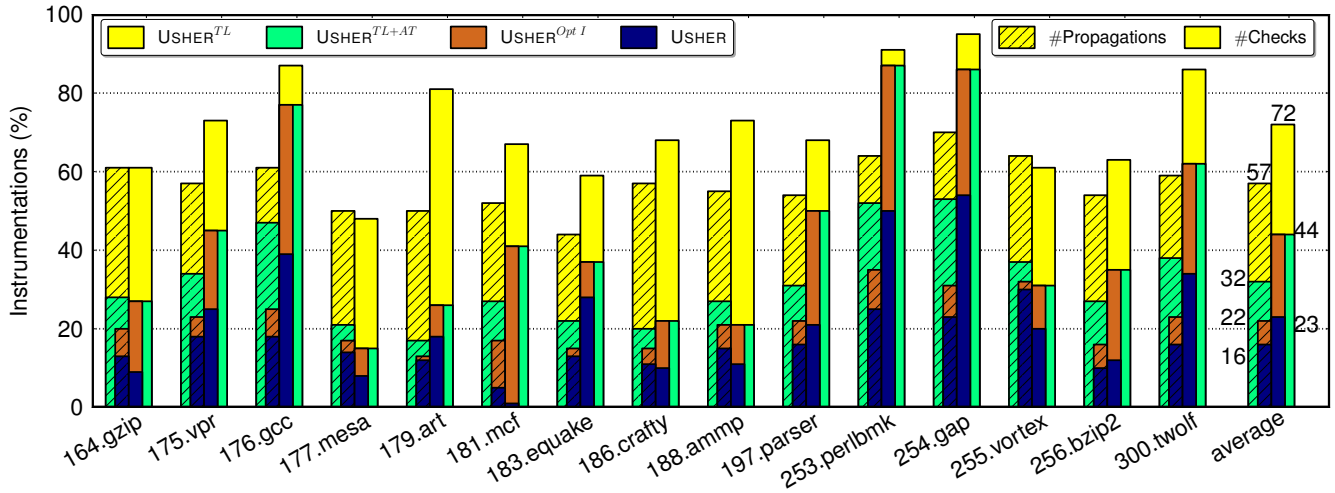


Figure 11: Static numbers of shadow propagations and checks performed at critical operations (normalized with respect to MSan).

to handle also address-taken variables, and (3) USHER^{OptI}, which is USHER^{TL+AT} extended to perform Opt I only. The average slowdowns for USHER^{TL}, USHER^{TL+AT} and USHER^{OptI} are 272%, 193% and 181%, respectively. One use of an undefined value is detected in the function *ppmatch()* of `197.parser` by all the analysis tools.

Figure 11 shows the static number of shadow propagations (i.e., reads from shadow variables) and the static number of runtime checks (at critical operations) performed by the four versions of our analysis (normalized with respect to MSan). USHER^{TL} can remove 43% of all shadow propagations and 28% of all checks performed by MSan, reducing its slowdown from 302% to 272%. By analyzing also address-taken variables, USHER^{TL+AT} has lowered this slowdown more visibly to 193%, by eliminating two-thirds of the shadow propagations and more than half of the checks performed by MSan. This suggests that a sophisticated value-flow analysis is needed to reduce unnecessary instrumentation for pointer-related operations. There are two major benefits. First, the flows of defined values from address-taken variables are now captured statically. Second, the statements that contribute no value flow to a critical operation do not need to be instrumented at all. However, the performance differences between USHER^{TL} and USHER^{TL+AT} are small for `253.perlbmk` and `254.gap`. For `253.perlbmk`, the majority (84%) of its VFG nodes reach a critical statement, where a runtime check is needed, as shown in Table 1. For `254.gap`, there are a high percentage (49%) of uninitialized address-taken variables when allocated and a relatively small number of strong updates (at 16%).

The two VFG-based optimizations bring further benefits to the USHER framework. Compared to USHER^{TL+AT}, USHER^{OptI} requires fewer shadow propagations, down from 32% to 22% on average, causing the slowdown to drop from 193% to 181%. If Opt II is also included, USHER can lower further the number of shadow propagations from 22% to 16% and the number of checks from 44% to 23%, resulting in an average slowdown of 123%. Due to Opt II, more nodes (10859 on average) are connected with $\widehat{\mathcal{T}}$, as shown in Figure 9. In an extreme case, `181.mcf` suffers from only a 2% slowdown. In this case, many variables that are used at frequently executed critical statements have received $\top$.

4.6 Effect of Compiler Optimizations on Reducing Instrumentation Overhead

We have also compared USHER and MSAN under their respective higher optimization settings, O1 and O2, even though this gives LLVM an opportunity to hide some uses of undefined values counter-productively [35], as discussed earlier. For an optimization level (O1 or O2) under both tools, a source file is optimized by (1) performing the LLVM optimizations at that level, (2) applying the USHER or MSAN analysis to insert the instrumentation code, and (3) rerunning the optimization suite at that level to further optimize the instrumentation code inserted.

MSAN and USHER suffer from 231% and 140% slowdowns, respectively, under O1, and 212% and 132%, respectively, under O2 on average. The best performer for both tools is `164.gzip`, with 104% (O1) and 102% (O2) for MSAN, and 26% (O1) and 20% (O2) for USHER. `255.vortex` is the worst performer for MSAN, with 501% (O1) and 469% (O2), and also for USHER under O1 with 300%. However, the worst performer for USHER under O2 is `253.perlbmk` with 288%. Note that USHER has higher slowdowns under O1 and O2 than O0+IM, since the base native programs benefit relatively more than instrumented programs under the higher optimization levels (in terms of execution times).

Therefore, USHER has reduced MSAN's instrumentation costs by 39.4% (O1) and 37.7% (O2) on average. Compared to O0+IM, at which USHER achieves an overhead reduction of 59.3% on average, the performance gaps have been narrowed when advanced compiler optimizations are enabled.

The users can choose different configurations to suit their different needs. For analysis performance, they may opt to O1 or O2 at the risk of missing bugs and having to decipher mysterious error messages generated. For debugging purposes, they should choose O0+IM.

5. RELATED WORK

5.1 Detecting Uses of Undefined Values

Prior studies rely mostly on dynamic instrumentation (at the binary or source-level level). The most widely used tool, Memcheck [24], was developed based on the Valgrind runtime instrumentation framework [21]. Recently, Dr. Memory [2], which is implemented on top of DynamoRIO [1, 36], runs twice as fast as Memcheck but is still an order of magnitude slower than the native code, although they both detect other bugs besides undefined memory uses. A few source-level instrumentation tools are also available, including Purify [12] and MSAN [9]. Source-level instrumentation can reap the benefits of compile-time optimizations, making it possible for MSAN to achieve a typical slowdown of 3X.

There are also some efforts focused on static detection [3, 14]. In addition, GCC and LLVM's clang can flag usage of uninitialized variables. However, their analysis are performed intraprocedurally, leading to false positives and false negatives. The problem of detecting uses of undefined values can also be solved by traditional static analysis techniques, including IFDS [23], typestate verification [8] and type systems [20] (requiring source-code modifications). However, due to its approximate nature, static analysis alone finds it rather difficult to maintain both precision and efficiency.

5.2 Combining Static and Dynamic Analysis

How to combine static and dynamic analysis has been studied for a variety of purposes. On one hand, static analysis can guide dynamic analysis to reduce its instrumentation overhead. Examples include taint analysis [5], buffer overflow attack protection [7], detection of other memory corruption errors [13] and WCET evaluation [19]. On the other hand, some concrete information about a program can be obtained at run time to improve the precision of static analysis. In [32], profiling information is used to guide source-level instrumentation by adding hooks to the identified contentious code regions to guarantee QoS in a multiple workload environment. In [26], dynamic analysis results are used to partition a streaming application into subgraphs, so that the static optimizations that are not scalable for the whole program can be applied to all subgraphs individually.

To detect uses of undefined values, a few attempts have been made. In [22], compile-time analysis and instrumentation are combined to analyze array-based Fortran programs, at 5X slowdown. Their static analysis is concerned with analyzing the definedness of arrays by performing a data-flow analysis interprocedurally. In [20], the proposed approach infers the definedness of pointers in C programs and checks those uncertain ones at run time. However, manual source code modification is required to satisfy its type system.

5.3 Value-Flow Analysis

Unlike data-flow analysis, value-flow analysis computes the def-use chains relevant to a client and puts them in some sparse representation. This requires the pointer/alias information to be made available by pointer analysis. Some recent studies improve precision by tracking value flows in pointer analysis [11, 16, 17], memory leak detection [29], program slicing [27] and interprocedural SSA analysis [4].

5.4 Pointer Analysis

Although orthogonal to this work, pointer analysis can affect the effectiveness of our value-flow analysis. In the current implementation of USHER, the VFG of a program is built based on the pointer information produced by an offset-based field-sensitive Andersen's pointer analysis available in LLVM [10]. To track the flow of values as precisely as possible, our value-flow analysis is interprocedurally flow-sensitive and context-sensitive. However, the presence of some spurious value-flow edges can reduce the chances for shadow values to be strongly updated. In addition, our context-sensitive definedness resolution may traverse some spurious value-flow paths unnecessarily, affecting its efficiency. So both the precision and efficiency of our value-flow analysis can be improved by using more precise pointer analysis [11, 16, 25, 28, 30, 31, 34] in future.

6. CONCLUSION

This paper introduces a new VFG-based static analysis, USHER, to speed up the dynamic detection of uses of undefined values in C programs. We have formalized and developed the first value-flow analysis framework that supports two flavors of strong updates to guide source-level instrumentation. Validation in LLVM using all the 15 SPEC2000 C programs demonstrates its effectiveness in significantly reducing the instrumentation overhead incurred by a state-of-the-art source-level dynamic analysis tool. In future work, we will focus on developing new VFG-based optimizations and new techniques for handling arrays and heap objects.

7. ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for their valuable comments. This work is supported by Australian Research Grants, DP110104628 and DP130101970, and a generous gift by Oracle Labs.

8. REFERENCES

[1] D. Bruening, T. Garnett, and S. P. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03*, pages 265–275, 2003.

[2] D. Bruening and Q. Zhao. Practical memory checking with Dr. Memory. In *CGO '11*, pages 213–223, 2011.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7), June 2000.

[4] S. Calman and J. Zhu. Increasing the scope and resolution of interprocedural static single assignment. In *SAS '09*, pages 154–170, 2009.

[5] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *CCS '08*, pages 39–50, 2008.

[6] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *CC '96*, 1996.

[7] W. Chuang, S. Narayanasamy, B. Calder, and R. Jhala. Bounds checking with taint-based analysis. In *HiPEAC '07*, pages 71–86, 2007.

[8] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA '06*, pages 133–144, 2006.

[9] Google. Memorysanitizer. `http://clang.llvm.org/docs/MemorySanitizer.html`, 2013.

[10] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *PLDI '07*, volume 42, pages 290–299, 2007.

[11] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO '11*, pages 289–298, 2011.

[12] R. Hastings and B. Joyce. Purify: fast detection of memory leaks and access errors. In *Winter 1992 USENIX Conference*, pages 125–138, 1991.

[13] J. Hiser, C. L. Coleman, M. Co, and J. W. Davidson. MEDS: the memory error detection system. In *ESSoS '09*, pages 164–179, 2009.

[14] R. Jiresal, A. Contractor, and R. Naik. Precise detection of un-initialized variables in large, real-life COBOL programs in presence of unrealizable paths. In *ICSM '11*, pages 448–456, 2011.

[15] C. Lattner and V. S. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–88, 2004.

[16] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *FSE '11*, pages 343–353, 2011.

[17] L. Li, C. Cifuentes, and N. Keynes. Precise and scalable context-sensitive pointer analysis via value flow graph. In *ISMM '13*, pages 85–96, 2013.

[18] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. In *CC '13*, pages 61–81, 2013.

[19] S. Mohan, F. Mueller, W. Hawkins, M. Root, C. A. Healy, and D. B. Whalley. ParaScale: exploiting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *RTSS '05*, pages 233–242, 2005.

[20] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[21] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07*, pages 89–100, 2007.

[22] T. V. N. Nguyen, F. Irigoin, C. Ancourt, and F. Coelho. Automatic detection of uninitialized variables. In *CC '03*, pages 217–231, 2003.

[23] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61, 1995.

[24] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX ATC '05*, pages 17–30, 2005.

[25] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO '12*, pages 264–274, 2012.

[26] R. Soulé, M. I. Gordon, S. P. Amarasinghe, R. Grimm, and M. Hirzel. Dynamic expressivity with static optimization for streaming languages. In *DEBS '13*, pages 159–170, 2013.

[27] M. Sridharan, S. J. Fink, and R. Bodík. Thin slicing. In *PLDI '07*, pages 112–122, 2007.

[28] Y. Sui, Y. Li, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. In *CGO '13*, pages 1–11, 2013.

[29] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA '12*, pages 254–264, 2012.

[30] Y. Sui, S. Ye, and J. Xue. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Softw. Pract. Exper.*, (To appear).

[31] Y. Sui, S. Ye, J. Xue, and P.-C. Yew. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. In *APLAS '11*, pages 155–171, 2011.

[32] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa. ReQoS: reactive static/dynamic compilation for QoS in warehouse scale computers. In *ASPLOS '13*, pages 89–100, 2013.

[33] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP '09*, pages 98–122, 2009.

[34] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO '10*, pages 218 – 229, 2010.

[35] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL '12*, pages 427–440, 2012.

[36] Q. Zhao, R. M. Rabbah, S. P. Amarasinghe, L. Rudolph, and W.-F. Wong. How to do a million watchpoints: efficient debugging using dynamic instrumentation. In *CC '08*, pages 147–162, 2008.