Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis

Yulei Sui, Ding Ye, and Jingling Xue, *Senior Member, IEEE*

Abstract—We introduce a static detector, SABER, for detecting memory leaks in C programs. Leveraging recent advances on sparse pointer analysis, SABER is the first to use a *full-sparse* value-flow analysis for detecting memory leaks statically. SABER tracks the flow of values from allocation to free sites using a sparse value-flow graph (SVFG) that captures def-use chains and value flows via assignments for all memory locations represented by both top-level and address-taken pointers. By exploiting field-, flow- and context-sensitivity during different phases of the analysis, SABER detects memory leaks in a program by solving a graph reachability problem on its SVFG. SABER, which is fully implemented in Open64, is effective at detecting 254 leaks in the 15 SPEC2000 C programs and seven applications, while keeping the false positive rate at 18.3 percent. SABER compares favorably with several static leak detectors in terms of accuracy (leaks and false alarms reported) and scalability (LOC analyzed per second). In particular, compared with FASTCHECK (which analyzes allocated objects flowing only into top-level pointers) using the 15 SPEC2000 C programs, SABER detects 44.1 percent more leaks at a slightly higher false positive rate but is only a few times slower.

Index Terms—Memory Leaks, sparse value-flow analysis, static analysis, pointer analysis

1 INTRODUCTION

THIS paper introduces the first static detector, SABER, which is fully implemented in the Open64 compiler, for detecting memory leaks in C programs by performing a full-sparse value-flow analysis. Table 1 compares SABER with several static memory leak detectors based on published and self-produced data on their accuracy in analyzing some or all of the 15 SPEC2000 C programs (totalling 620 KLOC). Table 2 focuses on scalability, i.e., LOC/sec taken in analyzing the programs handled by each detector. These speed numbers are rough estimates, since these detectors have been implemented in different programming languages (e.g., C++ and Java), executed on different machines and applied to different sets of programs.

Nevertheless, Tables 1 and 2 show that SABER provides a good tradeoff between scalability and accuracy in finding memory leaks with a low rate of false positives. These results, together with those reported later on analyzing seven applications (totalling 1.7 MLOC), show that SABER has met its design objectives and challenges, as discussed below.

Like these static detectors compared, SABER is not sound. Section 6.4 summarizes where and why SABER may fail to report certain leaks in a program.

1.1 Design Objectives

To find memory leaks statically in a C program, a leak analysis reasons about a *source-sink property*: every object created at an allocation site (a source) must eventually reach a free site (a sink) during any execution of the program. The analysis involves tracking the flow of values from sources to sinks through a sequence of memory locations represented by both top-level and address-taken pointers in the program. In order to be scalable and accurate, its underlying pointer analysis must also be scalable and accurate.

Current static detection techniques include ATHENA [13] (user specification), CONTRADICTION [22] (data-flow analysis), SATURN [33] (Boolean satisfiability), SPARROW [11] (abstract interpretation), CLANG [10] (symbolic execution) and FASTCHECK [3] (sparse value-flow analysis). Two approaches exist: *iterative data-flow analysis* and *sparse value-flow analysis*. The former tracks the flow of values iteratively at each point through the control flow while the latter tracks the flow of values sparsely through def-use chains or static single assignment (SSA) form. The latter is faster as the information is computed only where necessary using a sparse representation of value flows. Among all published static leak detectors, FASTCHECK is the only one in the latter category and all the others fall into the former category. However, FASTCHECK is limited to analyzing allocation sites whose values flow only into top-level pointers but ignores the remaining ones otherwise. Its sparse representation maintains precise def-use chains only for top-level pointers, which is obtained using the standard def-use analysis designed for scalars without the need to perform a pointer analysis.

Therefore, as shown in Tables 1 and 2, FASTCHECK is the fastest but not the most accurate. The other prior tools are significantly slower but can be more accurate as is the case for ATHENA, SATURN and SPARROW, because they reason about the flow of values through both top-level and address-taken pointers, albeit iteratively rather than sparsely.

This research draws its inspiration from the FASTCHECK work [3]. We aim to build SABER by using for the first time a full-sparse value-flow analysis for all memory locations. SABER tracks the flow of values from allocation to free sites using a sparse graph representation that captures def-use

• *The authors are with Programming Language and Compilers Group, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2035.*

Manuscript received 16 Jan. 2013; revised 12 Nov. 2013; accepted 15 Jan. 2014; date of publication 23 Jan. 2014; date of current version 4 Mar. 2014. Recommended for acceptance by M. Dwyer.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2014.2302311

TABLE 1
Comparing Accuracy in Analyzing the 15 SPEC2000 C Programs, Where "Fault Count" Is the Number of True Faults Detected

Leak Detector	SABER	
	Fault Counts	False Positive Rates (%)
ATHENA [13]	53 : 57	10 : 22
CONTRADICTION [22]	26 : 43	56 : 25
CLANG [10]	27 : 85	25 : 19
SPARROW [11]	81 : 77	16 : 17
FASTCHECK [3]	59 : 85	14 : 19

The data for the CLANG analyzer and SABER are from this paper while the data for the other tools are from the papers cited. Note that `perlbmk` is not analyzed by SPARROW because their "parser cannot accept many of its files", `gcc` is not analyzed by CONTRADICTION "because all warnings referred to data allocated via alloca", and `ammp`, `art`, `equake` and `mesa` are not analyzed by ATHENA. In each of the last two columns, SABER is compared with each detector D using the SPEC2000 C programs analyzed by D in the form of "D's data : SABER's data".

chains and value flows via assignments for both top-level and address-taken pointers. The edges in the graph are annotated with guards that represent branch conditions under which the value flow happens. Like FASTCHECK, SABER uses the guard information to reason about sink reachability on all paths. SABER is expected to be as accurate as SPARROW yet only slightly slower than FASTCHECK. This is feasible since full-sparse value-flow analysis can now be done more efficiently and accurately than before by leveraging recent advances on sparse pointer analysis [7], [8], [14], [31], [38].

1.2 Challenges

As shown in Tables 1 and 2, SPARROW is more accurate than FASTCHECK but a lot slower. To combine the best of both worlds, SABER needs to make a good balance between scalability and accuracy. SABER must be lightweight when reasoning about the flow of values from allocation sites through the def-use chains for address-taken pointers, which are ignored by FASTCHECK. In addition, such def-use chains must be accurate enough to allow more leaks to be detected. Finally, the false positive rate must be kept low.

1.3 Our Solution

SABER detects memory leaks using a full-sparse value-flow analysis. Top-level pointers do not require a pointer analysis to track the flow of values across them. However, for address-taken pointers, a pointer analysis is required due to the existence of indirect defs and uses through pointers. A traditional data-flow analysis computes the pointer information at every program point by respecting the control flow in the CFG of a program. This is costly as it propagates pointer information blindly from each node in the CFG to its successors without knowing if the information will be used there or not. In contrast, a sparse pointer analysis [7], [8], [31], [38] propagates the pointer information from variable defs directly to their uses along their def-use chains, but, unfortunately, the def-use information can only be computed using the pointer information. To break the cycle, a sparse pointer analysis typically proceeds in stages: def-use chains are initially approximated based on some fast pointer analysis and then refined in a sparse manner.

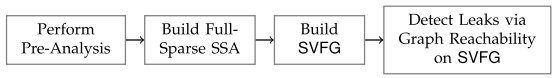

Fig. 1. Structure of the SABER detector.

SABER proceeds in four phases, as shown in Fig. 1. Their functionalities are described below, with details given in Sections 4.1, 4.2, 4.3, 4.4. To balance scalability and accuracy, SABER exploits field-, flow- and context-sensitivity during its analysis.

Phase 1: Pre-analysis. This is applied to the program to discover its pointer (and aliasing) information reasonably efficiently and accurately. To this end, we resort to flow- and context-insensitive Andersen's pointer analysis with offset-based field sensitivity and callsite-sensitive heap cloning for malloc wrappers.

Phase 2: Full-sparse SSA. This is built for each function individually, by considering all memory locations. We use a balanced model to represent the locations accessed indirectly at loads, stores and callsites. To improve accuracy, the pointer information obtained by pre-analysis is further refined sparsely with an intraprocedural flow-sensitive pointer analysis.

Phase 3: Sparse value-flow graph (SVFG). A sparse representation that captures def-use chains and value flows via assignments for all memory locations in the program, called a sparse value-flow graph, is constructed based on the full-sparse SSA form. Each def-use edge is annotated with a guard that captures the branch conditions between the def and the use in the

TABLE 2
Comparing Scalablity (LOC/Secs Taken) in Analyzing the SPEC2000 C Programs Handled by Each Detector, as Described in the Caption of Table 1

Leak Detector	Size (KLOC)	Speed (LOC/sec)	Machine (CPU and Memory)
ATHENA [13]	344	50	2.33 GHz Intel Xeon 4-core, 16GB memory
CONTRADICTION [22]	321	300	3.0 GHz Pentium 4, memory (size not given)
CLANG [10]	620	400	*Same as for* SABER
SPARROW [11]	465	284	3.2 GHz Pentium 4, 4GB memory
FASTCHECK [3]	671	37,900	3.2 GHz Pentium D, 3GB memory
SABER	620	10,220	3.0 GHz Intel Core2 Duo, 16 GB memory

The speed numbers for ATHENA, CONTRADICTION and SPARROW are calculated from the analysis times reported for the individual benchmarks in the papers cited. The speed number for FASTCHECK is reported for all the 15 SPEC2000 C programs and `bash` and `sshd` together. The speed numbers for CLANG and SABER are obtained in this paper.

TABLE 3
Six Types of Statements, Where p and q Are Local or Global Variables, v Is a Local or Global Variable, or a Heap Object, and k Uniquely Identifies a Callsite

Name	Syntax
Address	$p = \&v$
Copy	$p = q$
Load	$p = *q$
Store	$*p = q$
Call	$p = \mathscr{F}_k(\ldots, q, \ldots)$
Return	**return** p

program. Such guards are generated on-demand only when some allocation sites are analyzed during the leak detection phase.

Phase 4: Leak detection. This is performed by solving a graph reachability problem context-sensitively on the SVFG, starting from allocation sites (sources) and moving towards their reachable free sites (sinks).

The novelty lies in infusing field-sensitivity (by distinguishing different fields in a struct), flow-sensitivity (by tracking flow of statements) and context-sensitivity (by distinguishing different call sites of a function) at different phases of the analysis to balance scalability and accuracy judiciously.

1.4 Contributions

- SABER is the first that finds memory leaks by using a full-sparse value-flow analysis to track the flow of values through all memory locations and the first major client for demonstrating the benefits of sparse pointer analysis.
- SABER uses a new SVFG to maintain value flows for all memory locations, which may also be useful for other fault detection tools.
- SABER is effective at finding 254 leaks in the 15 SPEC2000 C programs and seven C applications while keeping the false positive rate at 18.3 percent.
- SABER compares favorably with several static leak detectors in terms of accuracy and scalability (as shown in Tables 1 and 2). In particular, compared with FASTCHECK (which analyzes allocated objects flowing only into top-level pointers) using the 15 SPEC2000 C programs, SABER detects 44.1 percent more leaks at a slightly higher false positive rate but is only a few times slower.

2 PROGRAM REPRESENTATION

SABER is designed to analyze fully-fledged C programs. The concepts of local variables, global variables, heap objects, pointers and memory locations are used in the standard manner. In particular, a memory location is identified by either a local variable or a global variable, or a heap object.

In the canonical form, as shown in Table 3, a statement in a C program is one of the following: (1) an assignment of the form, $p = \&v$ (address), $p = q$ (copy), $p = *q$ (load) or $*p = q$ (store), (2) a call statement, $p = \mathscr{F}_k(\ldots, q, \ldots)$, at callsite k, where $\mathscr{F}_k$ is understood to be a function pointer (or a function in the special case), and (3) a return statement, return p. Here, p and q are local or global variables and v is a local or global variable, or a heap object. Finally, each non-void function has a unique return statement.

During the conversion to SSA, three new types of operators are introduced: Φ, μ, and χ. The Φ functions are added at join points as is standard. In SSA form, each variable (or location) is defined exactly once in the program text. Distinct definitions of a variable are distinctly versioned. At a join point in the control-flow graph (CFG) of the program, all versions of the same variable reaching the point are combined using a Φ function, producing a new version for the variable.

Following [4], [6], potentially indirect uses (defs) at loads (stores) are identified by using μ (χ) functions. To enable the tradeoffs between efficiency and accuracy to be made, such indirectly accessed locations are represented by memory regions.

Definition 1 (Memory Regions). *A* (memory) region *is a set of memory locations that may be indirectly read (MAY-USE) or modified (MAY-DEF) at a statement.*

Each load $p = *q$ is annotated with a May-Use set, θ, which is a set of μ functions, such that every $\mu(R)$ denotes a region R that may be potentially read at the load. Similarly, each store $*p = q$ is annotated with a May-Def set, δ, which is a set of χ functions, such that every $R = \chi(R)$ indicates a region R that may be potentially defined at the store.

To understand this asymmetric treatment of μ and χ, suppose $R = \chi(R)$ (associated with $*p = q$) becomes $R_m = \chi(R_n)$ after SSA conversion. If p uniquely points to R, which must represent a single concrete memory location, then R_m can be strongly updated. R_m receives whatever q points to and the information in R_n is ignored. Otherwise, R_m must incorporate the pointer information from both R_n and q.

Every callsite is annotated with a May-Use set θ and a May-Def set δ to account for its interprocedural reference and modification side-effects, respectively. Every return statement in a function is annotated with a May-Use set θ to represent the set of regions that may be indirectly returned to a caller of the function.

When converted to SSA form using a standard algorithm, each $\mu(R)$ is treated as a use of R and each $R = \chi(R)$ is treated as both a def and use of R.

For a statement s, we will uniformly identify its may-use set θ and its may-def set δ as $[s]^{\theta}_{\delta}$. However, it is understood that (1) $\theta = \delta = \varnothing$ (for address and copy statements), (2) $\delta = \varnothing$ (for load and return statements), and (3) $\theta = \varnothing$ (for store statements).

3 A MOTIVATING EXAMPLE

We use an example in Fig. 2 to highlight why SABER can detect its two leaks with a full-sparse value-flow analysis while FASTCHECK can find only one of them. This example is adapted from a real scenario in `wine` in Fig. 13d (one of the applications used in our evaluation). In Fig. 2a, readBuf is called in a for loop in SerialReadBuf. Every time when readBuf is called, a single-char buffer formed by two objects

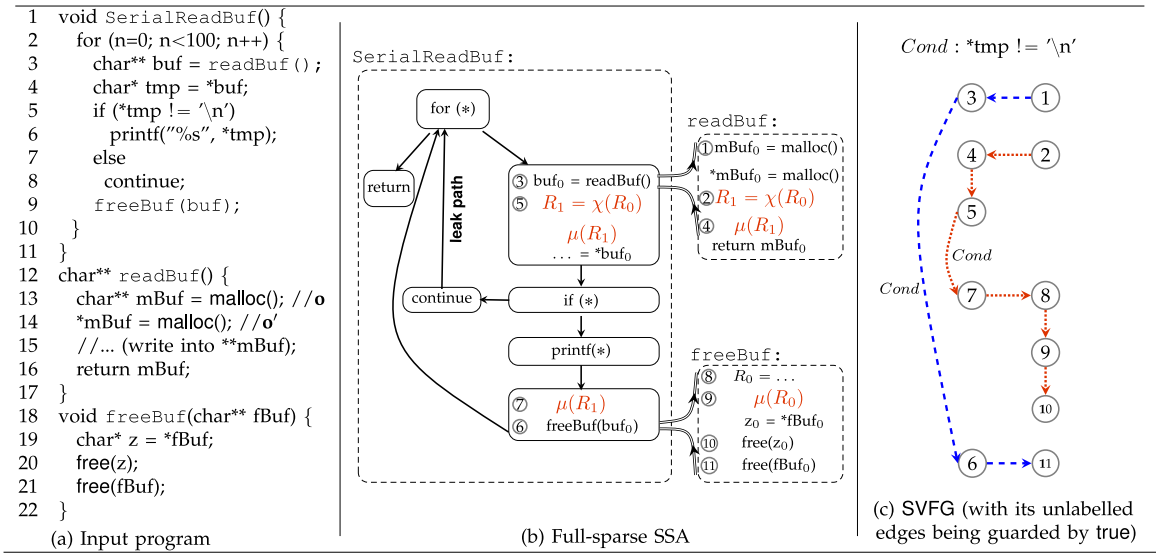

Fig. 2. A motivating example.

is created: o at line 13 and o' at line 14. There are two cases. If the buffer contains a char that is not `'\n'`, the char is printed and then both o and o' are freed. Otherwise, both objects leak.

We do not show how the flow of values is tracked into `*tmp`, i.e., into o' (with μ and χ functions), as this is irrelevant to the leak detection for o and o'.

Definition 2 (Points-to Sets). *For a pointer p, $ps(p)$ is a set of locations possibly pointed to by p*

Phase 1: Pre-analysis. We compute pointer information using (flow- and context-insensitive) Andersen-style analysis. The issues regarding field sensitivity and heap cloning (discussed in Section 4.1) are not relevant here. The following points-to sets are found:

$$ps(o) = \{o'\}$$
$$ps(\texttt{buf}) = ps(\texttt{mbuf}) = ps(\texttt{fbuf}) = \{\texttt{o}\}$$

Phase 2: Full-Sparse SSA. For this example, one region R is introduced to represent the singleton $\{o\}$, where o denotes an abstract heap object created at line 13. According to (1), R is aliased with `*buf`, `*mbuf` and `*fbuf`. Then the loads, stores, callsites and return statements in the program are annotated with μ and χ to make their indirect defs and uses explicit. For the store at line 14, $R = \chi(R)$ is added as R may be defined at the store. The loads at lines 4 and 19 are annotated with $\mu(R)$ since R may be read there. The callsite at line 3 is associated with $R = \chi(R)$ as R may be modified in `readBuf`. Similarly, $\mu(R)$ is added for the callsite at line 9 as R may be read in `freeBuf`. Finally $\mu(R)$ is added for the return at line 16 as R is implicitly returned to its callers. Note that $R_0 =$ is added in `freeBuf` as an implicit def as it receives its values from outside.

Fig. 2b gives the SSA form for each function derived in the standard manner.

Phase 3: SVFG. This can be built on the SSA form. As shown in Fig. 2c, the graph captures the def-use edges and value flows via assignments for o and o'.

Phase 4: Leak Detection. Proceeding similarly as in FastCheck [3], Saber checks if o and o' leak or not by solving a graph reachability on the SVFG separately in each case. Each def-use edge has a guard that captures branch conditions between the def and use in the interprocedural CFG of the program (given in Fig. 2b). All such guards are generated on-demand. As both o and o' reach a free site along the if-branch Cond $\equiv$ `*tmp != '\n'`. So Saber reports the leak warnings for both objects along the else branch.

FastCheck can find the leak of o but not o' since o flows into top-level pointers only but o' does not.

4 The Saber Detector

Saber detects memory leaks in a program by proceeding in the four phases given in Fig. 1. In this section, we describe these four phases in turn.

4.1 Pre-Analysis

Initially, we conduct a pre-analysis to compute the pointer information in a program reasonably quickly and accurately. We use Andersen's inclusion-based analysis, because it is the most precise among all flow- and context-insensitive pointer analyses and because it is scalable to millions of lines of code in minutes.

To improve precision further, our pre-analysis is offset-based field-sensitive. Different fields of a struct are distinguished. However, arrays are considered monolithic. Heap objects are modeled with context-sensitive heap cloning for allocation wrappers. All wrappers are identified and treated as allocation sites. Then the objects originating from an allocation site are represented by one single abstract object.

Allocation wrappers are detected similarly as in FastCheck [3]. A function f is marked as an allocator if f reveals a value flow from an allocation source to the return statement of f via a sequence of local variables. In our implementation, we make use of Open64's external function summaries to identify all allocation sources such as `malloc`.

After the pre-analysis, the points-to set $ps(v)$ for each pointer v is available. Each pointed-to target is either an abstract stack location (identified by a local or global variable) or an abstract heap object. The points-to sets in Fig. 2a are given in (1).

Since our pre-analysis is flow- and context-insensitive, SABER starts to exploit flow- and context-sensitivity from this point to improve its accuracy. To eliminate some spurious def-use chains for local variables in a function, we perform an intraprocedural sparse flow-sensitive pointer analysis to refine the points-to results.

4.2 Building Full-Sparse SSA Form

Our interprocedural full-sparse memory SSA is built modularly for each function. Top-level pointers are put into SSA just like scalars. To expose the memory locations indirectly accessed at loads, stores and callsites, we must ensure that the resulting sparse def-use chains are both accurate enough and amenable to fast traversal to satisfy the design objectives of SABER. For loads and stores, the indirect defs and uses are directly available after pre-analysis. For callsites, we perform a lightweight Mod-Ref analysis based again on the points-to information discovered at pre-analysis (Section 4.2.1). By performing both for a function, the memory regions accessed in (Section 4.2.2) and the SSA for (Section 4.2.3) the function are obtained.

Just like FASTCHECK [3] and SPARROW [11], SABER makes the following assumption.

Assumption 1. *Heap objects flowing (directly) into a global variable are considered not to leak.*

This is because an object pointed by a global variable may be used any time during program execution. How to find Java-style memory leaks (via global variables in C) is beyond the scope of this paper.

4.2.1 Performing the Mod-Ref Analysis

Definition 3 (Nonlocal Locations). *Consider a memory location ℓ that is not represented by a global variable but accessed in a function f. We say that ℓ represents a* local *location if (1) ℓ is locally declared in f and (2) f does not appear in any recursion cycle, and a* nonlocal *location otherwise. We write $Local_f$ ($NonLocal_f$) to represent the set of all local (nonlocal) locations accessed in f.*

Given a function f, the Mod-Ref analysis determines the set $\mathcal{U}$ ($\mathcal{D}$) of the nonlocal memory locations in f that may be indirectly read (modified) when f is executed, denoted as $\vdash f : \mathcal{U}, \mathcal{D}$. Due to Assumption 1, we do not track the side effects on global memory locations. During the analysis, each statement s in a function f, denoted as $\vdash f \downarrow s : \mathcal{U}, \mathcal{D}$, is analyzed individually. The rules used are given in Fig. 3. Our Mod-Ref analysis is sound in the sense that $\mathcal{U}$ and $\mathcal{D}$ in $\vdash f : \mathcal{U}, \mathcal{D}$ record all the nonlocal locations in $NonLocal_f$ read and modified by f, respectively.

The root causes for the interprocedural side-effects $\mathcal{U}$ and $\mathcal{D}$ in a function f are loads and stores. For a load $p = *q$, $ps(q)$ may contain nonlocal locations read in f ([I-LD]). Similarly, [I − ST] collects nonlocal locations in $ps(q)$ that may be modified at a store. In contrast, address, copy and return statements do not contribute any side effects according to [I-ADD], [I-COPY] and [I-RET].

$$\text{[I-ADD]}\ \frac{}{\vdash f \downarrow p = \&v : \varnothing, \varnothing}$$

$$\text{[I-COPY]}\ \frac{}{\vdash f \downarrow p = q : \varnothing, \varnothing}$$

$$\text{[I-RET]}\ \frac{}{\vdash f \downarrow \textbf{return}\ p : \varnothing, \varnothing}$$

$$\text{[I-LD]}\ \frac{\mathcal{U} = \{v \mid v \in ps(q) \wedge v \in NonLocal_f\}}{\vdash f \downarrow p = *q : \mathcal{U}, \varnothing}$$

$$\text{[I-ST]}\ \frac{\mathcal{D} = \{v \mid v \in ps(p) \wedge v \in NonLocal_f\}}{\vdash f \downarrow *p = q : \varnothing, \mathcal{D}}$$

$$\text{[I-PROC]}\ \frac{\begin{array}{c} s_1, \ldots, s_n \text{ are the statements in } f \\ \vdash f \downarrow s_i : \mathcal{U}_i, \mathcal{D}_i \\ \mathcal{U} = (\bigcup_i \mathcal{U}_i) \backslash Local_f \\ \mathcal{D} = (\bigcup_i \mathcal{D}_i) \backslash Local_f \end{array}}{\vdash f : \mathcal{U}, \mathcal{D}}$$

$$\text{[I-CALL]}\ \frac{\begin{array}{c} Callees(\mathscr{F}_k) = \{g_1, \ldots, g_n\} \quad \vdash g_i : \mathcal{U}_i, \mathcal{D}_i \\ N_{\mathscr{F}_k} \text{ is the set of nonlocal locations accessed} \\ \text{at } \mathscr{F}_k \text{ and visible inside } f \\ \mathcal{U} = N_{\mathscr{F}_k} \cap \bigcup_i \mathcal{U}_i \quad \mathcal{D} = N_{\mathscr{F}_k} \cap \bigcup_i \mathcal{D}_i \end{array}}{\vdash f \downarrow \_ = \mathscr{F}_k(\_) : \mathcal{U}, \mathcal{D}}$$

Fig. 3. The Mod-Ref analysis.

For a callsite $\mathscr{F}_k$, the most conservative Mod-Ref analysis is to assume that the set of all nonlocal locations passed into this callsite, $N_{\mathscr{F}_k}$, may be read and modified by its callees invoked directly/indirectly. The notation $Callees(\mathscr{F}_k)$ denotes the set of callees possibly invoked at the callsite $\mathscr{F}_k$. This naive approach is inaccurate since an overwhelmingly large number of unrealizable def-use chains would be created across the functions. Instead, we perform a refined analysis for the callees invoked at $\mathscr{F}_k$ to filter out spurious Mod-Ref information. In the presence of recursion, [I-CALL] and [I-PROC] are recursively applied until a fixed point is reached. Note that in [I-PROC], all local locations in $Local_f$ introduced by analyzing a callsite in f are removed.

4.2.2 Generating Memory Regions

There are many solutions depending on how the memory is partitioned. At one extreme, FASTCHECK [3] assumes that all dereference expressions are essentially aliased with one special region. This coarsest partitioning makes it fast but too inaccurate to analyze allocation sites whose values flow into this special region. At the other extreme, distinct locations in $ps(v)$ for a pointer v are distinct regions aliased with $*v$. This finest partitioning would make an analysis accurate but traverse too many def-use chains to be efficient.

SABER adopts a balanced memory model to partition the locations accessed in a function according to the points-to information at loads and stores and the Mod-Ref information at callsites.

By Definition 1, a memory region is a set of memory locations indirectly accessed. In SABER, the set of memory regions, denoted $\mathbb{R}_f$, for a function f is created as follows. There are three kinds of regions, formed based on $Local_f$ and $NonLocal_f$ (Definition 3):

$$[\texttt{S-ADD}]\ \frac{[p = \&v]^{\theta}_{\delta}}{\theta = \delta = \varnothing}$$

$$[\texttt{S-COPY}]\ \frac{[p = q]^{\theta}_{\delta}}{\theta = \delta = \varnothing}$$

$$[\texttt{S-LD}]\ \frac{[p = *q]^{\theta}_{\delta} \quad R \in \mathbb{R}_f \quad R \cap ps(q) \neq \varnothing}{\theta \cup= \{\mu(R)\} \quad \delta = \varnothing}$$

$$[\texttt{S-ST}]\ \frac{[*p = q]^{\theta}_{\delta} \quad R \in \mathbb{R}_f \quad R \cap ps(p) \neq \varnothing}{\theta = \varnothing \quad \delta \cup= \{R = \chi(R)\}}$$

$$[\texttt{S-CSMU}]\ \frac{\begin{array}{ll}[\_ = \mathscr{F}_k(\_)]^{\theta}_{-} & \vdash \mathscr{F}_k : \mathcal{U}, \mathcal{D} \\ R \in \mathbb{R}_f & R \cap \mathcal{U} \neq \varnothing\end{array}}{\theta \cup= \{\mu(R)\}}$$

$$[\texttt{S-CSCHI}]\ \frac{\begin{array}{ll}[\_ = \mathscr{F}_k(\_)]^{-}_{\delta} & \vdash \mathscr{F}_k : \mathcal{U}, \mathcal{D} \\ R \in \mathbb{R}_f & R \cap \mathcal{D} \neq \varnothing\end{array}}{\delta \cup= \{R' = \chi(R')\}}$$

$$[\texttt{S-RET}]\ \frac{[\textbf{return}\ \_]^{\theta}_{\delta} \quad \vdash f : \mathcal{U}, \mathcal{D} \quad R \in \mathbb{R}_f \quad R \cap \mathcal{D} \neq \varnothing}{\theta \cup= \{\mu(R)\} \quad \delta = \varnothing}$$

Fig. 4. Annotating the may-uses and may-defs at the statements of a function f with μ and χ functions.

Global Region. All global variables in the program are represented by one *GLOBAL* region, as also in FASTCHECK [3] and SPARROW [11]. Due to Assumption 1, heap objects flowing into *GLOBAL* are assumed not to leak.

Nonlocal Regions. For every $*p$ at a load $\cdots = *p$ or a store $*p =$, a nonlocal region $\{v \mid v \in ps(p) \wedge v \in NonLocal_f\}$ is created. For a callsite $\mathscr{F}_k$, where $\vdash \_ = \mathscr{F}_k(\_) : \mathcal{U}, \mathcal{D}$, a nonlocal region $\{v\}$, where $v \in NonLocal_f$, is created for every variable $v \in (\mathcal{U} \cup \mathcal{D})$.

Local Regions. Every local variable $v \in Local_f$ is in its own local region, $\{v\}$. For every $*p$ at a load $= *p$ or a store $*p = \dot{s}$, a local region $\{v\}$, where $v \in Local_f$ is created for every $v \in ps(p)$. For a callsite $\mathscr{F}_k$, where $\vdash \_ = \mathscr{F}_k(\_) : \mathcal{U}, \mathcal{D}$, a local region $\{v\}$, where $v \in Local_f$, is created for every $v \in (\mathcal{U} \cup \mathcal{D})$.

It is understood that $\mathbb{R}_f$ never contains a region R that is a strict subset of $R' \in \mathbb{R}_f$. In this case, R is redundant (and should be removed) since an access to R' always implies an access to R.

We have modeled the memory this way because it provides a good tradeoff between scalability and accuracy as validated in our experiments.

4.2.3 Constructing the SSA

Once all regions in a function are identified, may-defs and may-uses at its loads, stores, callsites and its return statement are exposed by adding μ and χ functions and then the conversion to SSA can take place using a standard SSA algorithm [4], [6].

The detailed rules are given in Fig. 4. Recall from Section 2 that $[s]^{\theta}_{\delta}$ indicates that statement s is associated with a may-use set θ and a may-def set δ. In order to initialize a nonlocal region R initially accessed in a function f, a dummy statement of the form "$R = s$." is introduced at its entry to serve as its initial def. After the SSA conversion is done, R at the left-hand side has version 0 as it expects to "receive" values from its callers. This is illustrated using freeBuf in Fig. 2b. As address and copy statements do not have may-defs and may-uses, there is no need to insert any μ or χ ([S-ADD] and [S-COPY]). For loads and stores with indirectly accessed regions, aliases are recognized as overlapping regions and accounted for explicitly by adding μ functions ([S-LD]) and χ functions ([S-ST]). Similarly, given the Mod-Ref information at a callsite, aliased regions are inserted in the form of μ and χ functions to represent may-uses ([S-CSMU]) and may-defs ([S-CSCHI]). Finally, a $\mu(R)$ function is added to the return statement of f for every nonlocal region R that may be modified since R may be implicitly returned to its callers ([S-RET]).

Let us revisit our example given in Fig. 2a. Based on (1), only one nonlocal region relevant to leak detection is found for every function in the program: $R = \{o\}$. Thus, the SSA form for the program is obtained as shown in Fig. 2b.

Let us explain the rules in Fig. 4 by considering one more example in Fig. 5. There are two functions, `foo` and `bar`. We focus on analyzing `foo` with and without a call to `bar`. In `foo`, p, q, v and w are local locations while x, y and z are nonlocal locations accessed (Fig. 5a). The relevant points-to sets found by pre-analysis are given in Fig. 5b. There are four regions created in Fig. 5c by proceeding as described in Section 4.2.2. The notation $R(v_1, \ldots, v_n)$ stands for a region R that represents variables $v_1, \ldots, v_n$. For `bar`, which is omitted, we are only interested in its side-effects as assumed in Fig. 5d when it is called in `foo`.

Let us first consider `foo` without a call to `bar` as illustrated Fig. 5e. For the load $\cdots = *p$, $*p$ is aliased with $R(v), R(x, y)$ and $R(x, z)$. So the three μ functions are inserted ([S-LD]). For the store $*q = \cdots$, the three χ functions are inserted by applying [S-ST]. As $R(x, y)$ and $R(x, z)$ are the nonlocal regions modified in `foo`, they are associated in the form of μ functions with the return statement of `foo` ([S-RET]).

Let us move to Fig. 5f when `foo` contains a callsite for `bar`, with its Mod-Ref information assumed as in Fig. 5d. Since $\mathcal{U} = \{v\}$ and $\mathcal{D} = \{y\}$, two nonlocal regions, $R(v)$ and $R(y)$, are created at this callsite. $R(v)$ already exists since it was created earlier in Fig. 5c. However, $R(y)$ is redundant and thus removed since $R(y)$ is a subset of $R(x, y)$ created earlier. $R(v)$ is aliased with $\mathcal{U}$ and $R(x, y)$ is aliased with $\mathcal{D}$. The aliased regions are inserted in the form of μ and χ for the callsite ([S-CSMU] and [S-CSCHI]). In both Figs. 5e and 5f, the function `foo` can be put into SSA in the standard manner.

4.3 Building SVFG

Once a function is in SSA, the def-use chains in the function are available, but these are insufficient for SABER to check leaks caused interprocedurally. In this section, we describe how to build our SVFG to capture def-use chains and value flows by assignments within and across the procedural boundaries.

The SVFG of a program is kept simple. The only statements reachable directly or indirectly from all allocation sites being analyzed need to be considered. Its nodes represent variable definitions, except for one caveat regarding indirect uses added as μ functions to a callsite explained

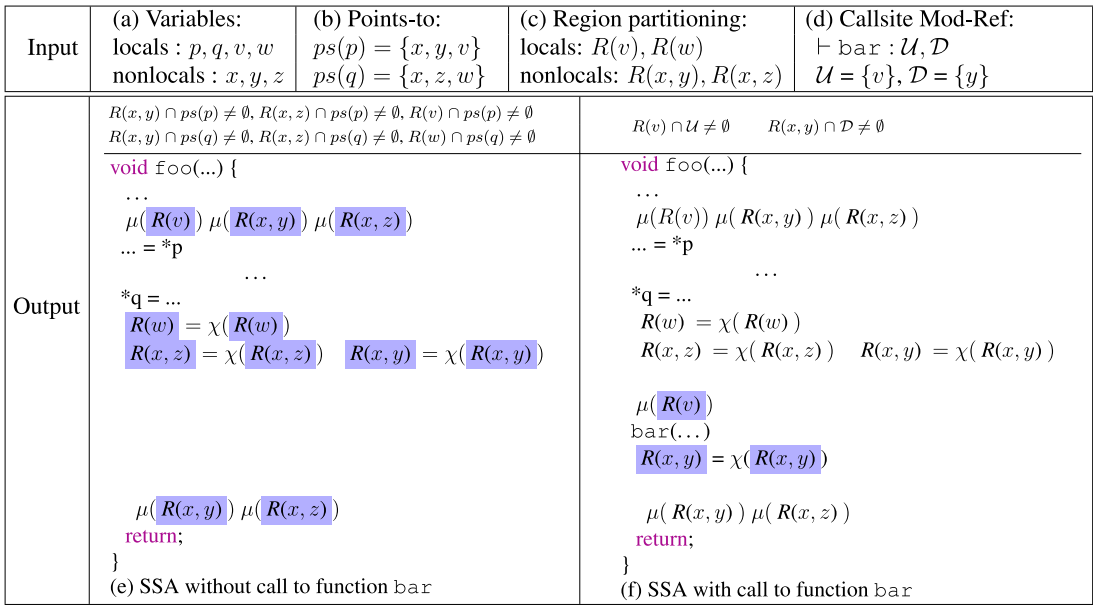

Fig. 5. An example for illustrating the construction of SSA for address-taken pointers.

below. After SSA conversion, a region R is split into multiple versions as R_i, where $i \geqslant 0$, We write $\widehat{R_i}$ for the def site of a memory region R with version i. For a local variable p_i, which forms a region by itself, the def site is often denoted $\widehat{p_i}$ for brevity. For a global variable g_i, which is abstracted as the unique *GLOBAL* region (Section 4.2.2), its def site is $\widehat{g_i}$, which is equivalent to $\widehat{GLOBAL}$.

The bulk of the task involved in building the SVFG lies in adding its edges to capture def-use chains and value flows via assignments. The flow of values, denoted between a pair of definitions $\widehat{R_i} \overset{k}{\longleftarrow} \widehat{R_j}$, indicates that the values flow from the def site of region R_j to the def site of R_i, where k represents a callsite id for context-sensitive reachability analysis and is omitted if the flow is intraprocedural.

The rules used are given in Fig. 6, with each rule illustrated by an example in Fig. 7. By applying [V-COPY], [V-MU], [V-CHI] and [V-PHI] to a function, its intraprocedural def-use chains are added. In [V-COPY], instead of linking $\widehat{q_j}$ to the use q_j at a copy statement and then linking the use to $\widehat{p_i}$, we add one single edge $\widehat{p_i} \leftarrow \widehat{q_j}$ directly. We do the same in the other rules. In [V-MU], for each $\mu(R_t)$ function associated with a load $p_i = *q_j$, the value flows from the def site $\widehat{R_t}$ to $\widehat{p_i}$. In [V-CHI] for a store, $\widehat{R_t} \leftarrow \widehat{R_s}$ signifies a weak update to region R_t using the old information in R_s and can be ignored if a strong update is possible (as described in Section 2). For example, in Fig. 6c, the value flow from line 2 to line 4 is invalid since p_1 points to a single, concrete location (if foo does not appear a recursion cycle). For a Φ operation, the value flows from the two branches into the def site at the left-hand side ([V-PHI]).

There are three rules for dealing with interprocedural value flow; these rules look complex but are also conceptually simple. An allocation site is marked as a *source* and a free site as a *sink*. As before, $Callees(\mathscr{T}_k)$ stands for a mapping from $\mathscr{T}_k$ to a set of callees. [V-CSPAR] captures the value flows for the standard parameter passing and return for top-level pointers. In this rule, $PAR_g(q_j)$ stands for the corresponding formal parameter of q_j in SSA (version 0) and $RET_g(p_i)$ identifies the unique SSA variable returned in function g, where $g \in Callees(\mathscr{T}_k)$. [V-CSCHI] accounts for the "implicit" value returns for address-taken pointers. $\widehat{R_t} \leftarrow \widehat{R_s}$ in this rule is needed just like the case for [V-CHI] if a weak update on R_t is performed so that the old points-to information in R_s is preserved. Similarly, [V-CSMU] models the "implicit" parameter passing for address-taken pointers by treating the site of $\mu(R_t)$, denoted by $\widehat{\mu(R_t)}$, as a pseudo formal parameter (def) site. This is crucial because we must record the control-flow paths under which the value flow happens in order to reason about memory leaks interprocedurally.

$$\text{[V-COPY]}\ \frac{[p_i = q_j]^-_-}{\widehat{p_i} \longleftarrow \widehat{q_j}}$$

$$\text{[V-MU]}\ \frac{[p_i = *q_j]^\theta_- \quad (\mu(R_t)) \in \theta}{\widehat{p_i} \longleftarrow \widehat{R_t}}$$

$$\text{[V-CHI]}\ \frac{[*p_i = q_j]^-_\delta \quad (R_t = \chi(R_s)) \in \delta}{\widehat{R_t} \longleftarrow \widehat{R_s} \quad \widehat{R_t} \longleftarrow \widehat{q_j}}$$

$$\text{[V-PHI]}\ \frac{p_i = \Phi(q_j, q_k)}{\widehat{p_i} \longleftarrow \widehat{q_j} \quad \widehat{p_i} \longleftarrow \widehat{q_k}}$$

$$\text{[V-CSPAR]}\ \frac{[p_i = \mathscr{F}_k(\ldots, q_j, \ldots)]^-_- \quad g \in Callees(\mathscr{F}_k)}{\widehat{PAR_g(q_j)} \overset{(^g_k}{\longleftarrow} \widehat{q_j} \quad \widehat{p_i} \overset{)^g_k}{\longleftarrow} \widehat{RET_g(p_i)}}$$

$$\text{[V-CSMU]}\ \frac{[\_ = \mathscr{F}_k(\_)]^\theta_\delta \quad g \in Callees(\mathscr{F}_k) \quad \mu(R_t) \in \theta}{U^g_k(R_t) \overset{(^g_k}{\longleftarrow} \widehat{\mu(R_t)} \quad \widehat{\mu(R_t)} \longleftarrow \widehat{R_t}}$$

$$\text{[V-CSCHI]}\ \frac{[\_ = \mathscr{F}_k(\_)]^\theta_\delta \quad g \in Callees(\mathscr{F}_k) \quad R_t = \chi(R_s) \in \delta}{\widehat{R_t} \overset{)^g_k}{\longleftarrow} D^g_k(R_t) \quad \widehat{R_t} \longleftarrow \widehat{R_s}}$$

Fig. 6. Rules used for building SVFG.

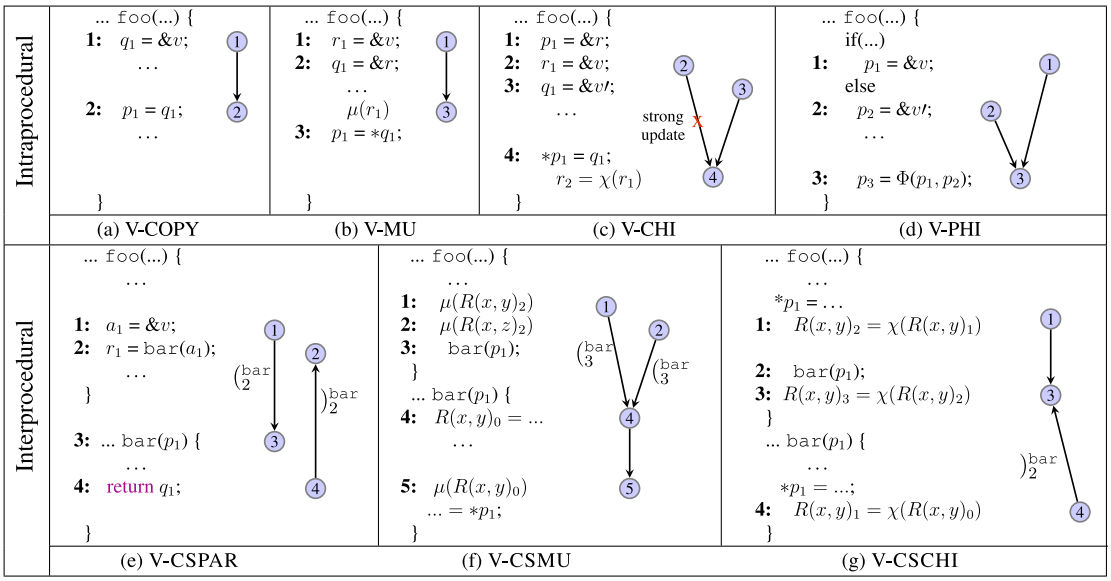

Fig. 7. Examples for illustrating the seven rules given in Fig. 6.

Let us now take a closer look at the technical details behind [V-CSMU] and [V-CSCHI]. Given a function $\vdash g : \mathcal{U}; \mathcal{D}$, we can apply the rules in Fig. 3 to build $ENT_g = \{R \in \mathbb{R}_g \mid R \cap \mathcal{U} \neq \varnothing\}$, which denotes the set of nonlocal regions that may be read in g, and $EXT_g = \{R \in \mathbb{R}_g \mid R \cup \mathcal{D} \neq \varnothing\}$, which denotes the set of nonlocal regions that may be modified or the regions returned to a caller at the exit of g (captured by [S-RET]). By construction, each SSA variable in ENT_g has version 0 as it expects to "receive" values from a caller of g. Similarly, each SSA variable in EXT_g has the largest version for the underlying variable as it contains the final value defined in g. For freeBuf in Fig. 2, $ENT_{\texttt{freebuf}} = \{R_0\}$. The def "$R_0 = \cdots$" added in freeBuf serves to receive its values from outside. For readBuf, we have $EXT_{\texttt{readbuf}} = \{R_1\}$.

Given a $\mu(R_t)$ at callsite $\_ = \mathcal{F}_k(\_)$, for each callee g, we define $U_k^g(R_t) = \{R \in ENT_g \mid R \cap \widehat{R_t} \neq \varnothing\}$. We regard $\mu(R_t)$ as a pseudo formal parameter, $\mu(\widehat{R_t})$, so that $\widehat{R_t}$ is first propagated to $\mu(\widehat{R_t})$ and then to each region in $U_k^g(R_t)$, by [V-CSMU]. This realizes the implicit parameter passing for address-taken pointers. Given a $R_t = \chi(R_s)$, we define $D_k^g(R_t) = \{R \in EXT_g \mid R \cap R_t \neq \varnothing\}$ to identify all regions in EXT_g that alias with region R_t. By [V-CSCHI], an edge is added from every region in $D_k^g(R_t)$ to R_t (to realize the implicit value returns for address-taken pointers).

To achieve context-sensitive reachability analysis during leak detection, call and return edges are labelled with call-site information in the standard manner. In Rules [V-CSPAR], [V-CSMU] and [V-CSCHI], the call edges are labelled with the open parenthesis $(_k^g$ and the return edges with the close parenthesis $)_k^g$. During leak detection, realizable interprocedural value flows correspond to paths containing properly nested parentheses and context-sensitivity is achieved by solving a context-free language (CFL) reachability problem as demonstrated in [16], [23], [26], [27], [28].

Let us see how the SVFG in Fig. 2c is built. The part corresponding to the source ① tracks the flow of o through the top-level pointers only into the sink ⑪, as is the case in FastCheck [3]. The other part that tracks the flow of o' through the address-taken pointers, starting from the source ② and ending at the sink ⑩. Its edges are constructed as follows: ④ ← ② by [V-CSCHI], ⑦ ← ⑤ ← ④ by [V-CSMU], ⑧ ← ⑦ by [V-MU], and ⑨ ← ⑧ by [V-CSPAR].

4.4 Leak Detection

Once the SVFG is built, the guards on its edges are computed on-demand to capture path conditions under which the value flow happens in the program. The guard information is used to reason about sink reachability on all paths. Saber proceeds similarly as FastCheck except that Saber uses Binary Decision Diagrams (BDDs) to encode paths while FastCheck uses a SAT solver to reason about them.

Given a source object, src, created at an allocation site, the sink reachability algorithm proceeds in the following two stages:

Some Path Analysis. We find the set of nodes, denoted $\mathcal{F}_{src}$ and called a *forward slice*, reachable from src in the SVFG. This is context-sensitive by matching call and return edges to rule out unrealizable interprocedural flows of values [23], [27], [28]. Let $\mathcal{S}_{src}$ be the set of sinks, i.e., free sites reached in $\mathcal{F}_{src}$. If $\mathcal{S}_{src} = \varnothing$, then src definitely leaks. In this case, src is known not to reach a sink along some control-flow paths. If src reaches *GLOBAL* along some control-flow paths, the leak detection phase stops (for src), assuming that src does not leak (Assumption 1).

All Path Analysis. We refine $\mathcal{F}_{src}$ into a *backward slice*, denoted $\mathcal{B}_{src}$, that consists of only nodes on paths connecting src to a sink in $\mathcal{S}_{src}$. Then we perform an all-path analysis to check that src reaches at least a sink in $\mathcal{S}_{src}$ on every control-flow path that src flows to. We report a leak warning if src does not reach a sink in $\mathcal{S}_{src}$ on some (one or more) control-flow paths. Such faults are called *conditional leaks*.

We now describe how to solve our all-path graph reachability problem. For a sink tgt in $\mathcal{S}_{src}$, let $vfpaths(src, tgt)$ be the set of all *value-flow paths* from src to tgt in the SVFG.

Recursion is bounded so that recursive callsites are invoked at most once. For each value-flow path $P \in vfpaths(src, tgt)$, $vfguard(P)$ is a Boolean formula that encodes the set of *control-flow paths* that the underlying value reaches in the program, from src to tgt. By convention, `true` denotes the set of all control-flow paths between a pair of points. Like recursion, loops are bounded to at most one iteration. Thus, we have:

$$\mathsf{FREED}(src) = \bigvee_{tgt \in \mathcal{S}_{src}} \bigvee_{P \in vfpaths(src,tgt)} vfguard(P) \quad (2)$$

which signifies the set of control-flow paths reaching a sink in $\mathcal{S}_{src}$ from src. If $\mathsf{FREED}(src) \not\equiv \mathsf{true}$, a leak warning is issued, indicating that src leaks along the control-flow paths specified by $\neg\mathsf{FREED}(src)$.

To compute $vfguard(P)$, let $vfedges(P)$ be the set of all value-flow edges in P. For each $(\widehat{p}, \widehat{q}) \in vfedges(P)$, we write $cfguard(\widehat{p}, \widehat{q})$ as a Boolean formula to represent the set of control-flow paths in the program on which the def $\widehat{p}$ reaches its use at the def $\widehat{q}$ site, denoted by $cfpaths(\widehat{p}, \widehat{q})$. Thus, we have:

$$vfguard(P) = \bigwedge_{(\widehat{p}, \widehat{q}) \in vfedges(P)} cfguard(\widehat{p}, \widehat{q}). \quad (3)$$

There are two cases. If $(\widehat{p}, \widehat{q})$ is a call or return edge as marked in [CS − CSPAR], [CS − CSMU] and [CS − CSCHI] in Fig. 6, then $cfguard(\widehat{p}, \widehat{q}) = \mathsf{true}$ trivially since $|cfpaths(\widehat{p}, \widehat{q})| = 1$. Otherwise, $\widehat{p}$ and $\widehat{q}$ are two program points in the same function. There can be many control-flow paths in $cfpaths(\widehat{p}, \widehat{q})$. Let each path $Q \in cfpaths(\widehat{p}, \widehat{q})$ be uniquely identified by $pguard\ (Q)$. Thus, we have:

$$cfguard(\widehat{p}, \widehat{q}) = \bigvee_{Q \in cfpaths(\widehat{p}, \widehat{q})} pguard(Q). \quad (4)$$

To compute $pguard(Q)$, we assign a Boolean condition to every edge in a CFG. If a node is a branch point, a unique Boolean guard C is generated. The true branch is assigned C and the false branch $\neg C$. Otherwise, the unique outgoing edge is given true. Let $E(Q)$ be the set of all edges in Q and $eguard(e)$ the guard on edge $e \in E(Q)$. Finally, we have:

$$pguard(Q) = \bigwedge_{e \in E(Q)} eguard(e). \quad (5)$$

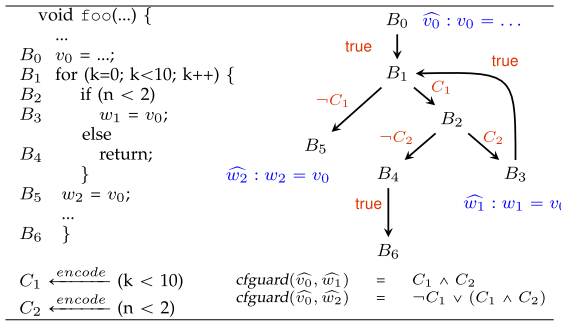

Fig. 8. Encoding paths with Boolean guards.

There is one caveat as illustrated in Fig. 8 using a portion of a backward slice regarding how the exit of a loop is handled. Note that $cfguard(\widehat{v_0}, \widehat{w_1}) = C_1 \wedge C_2$ because there is only one path from $\widehat{v_0}$ to $\widehat{w_1}$: $Q := B_0 \xrightarrow{\mathsf{true}} B_1 \xrightarrow{C_1} B_2 \xrightarrow{C_2} B_3$. So $cfguard(\widehat{v_0}, \widehat{w_1}) = pguard(Q) = C_1 \wedge C_2$. To compute $cfguard\ (\widehat{v_0}, \widehat{w_2})$, there are two paths to consider: (1) $Q_1 := B_0 \xrightarrow{\mathsf{true}} B_1 \xrightarrow{\neg C_1} B_5$ and (2) $Q_2 := B_0 \xrightarrow{\mathsf{true}} B_1 \xrightarrow{C_1} B_2 \xrightarrow{C_2} B_3 \xrightarrow{\mathsf{true}} B_1 \xrightarrow{\neg C_1} B_5$. Recall that the analysis bounds a loop to at most one iteration. For Q_2, entering the loop is described by C_1 but exiting it is by $\neg C_1$. As in FASTCHECK, SABER drops the exit condition $\neg C_1$ in order to make Q_2 feasible. Thus, $pguard(Q_1) = \neg C_1$ and $pguard(Q_2) = C_1 \wedge C_2$. Finally, $cfguard(\widehat{v_0}, \widehat{w_2}) = pguard(Q_1) \vee pguard(Q_2) = \neg C_1 \vee (C_1 \wedge C_2)$.

Consider Fig. 2c. There are two objects o and o' to be analyzed. For each source, there is one value-flow path connecting it to one sink. $Cond$ represents the guard assigned to the true if-branch. So $cfguard(③, ⑥) = cfguard(④, ⑤) = Cond$, capturing the branch condition in each case. Thus, $\mathsf{FREED}(o) = \mathsf{FREED}(o') = Cond$. So both are considered to leak on paths $\neg Cond$ since $Cond \not\equiv \mathsf{true}$.

5 THE SABER IMPLEMENTATION

We have implemented SABER in Open64, an open-source industry-strength compiler, at its interprocedural analysis (IPA) phase, as shown in Fig. 9. IPA performs global

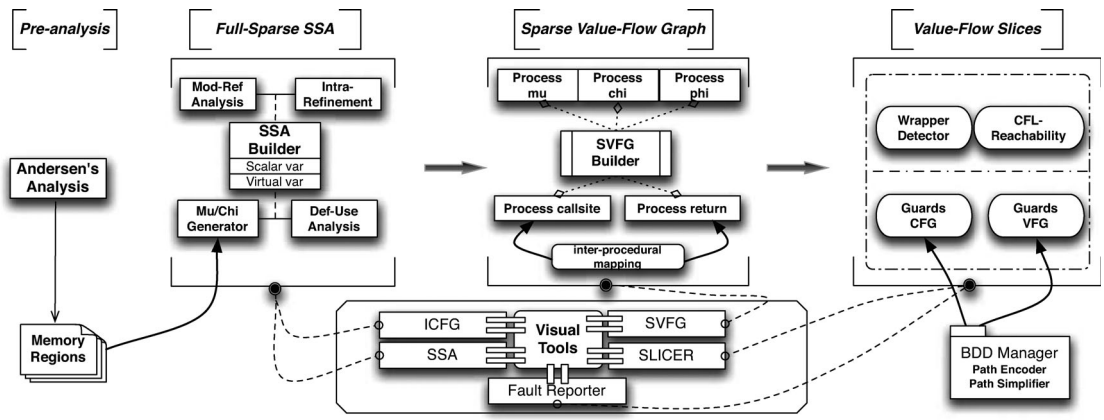

Fig. 9. An implementation of SABER in the Open64 compiler.

analysis by combing information from its IPL (Local part of its Interprocedural phase, which collects summary information local to a function). SABER operates on its High WHIRL intermediate representation, which preserves high-level control flow constructs, such as DO_LOOP and IF, and is ideal for value-flow analysis. In the latest Open64 release, its WHIRL SSA form is still intraprocedural and used mainly to support intraprocedural optimizations. We have extended it by using the Alias Tags provided in Open64 to represent memory regions, thereby obtaining an SVFG for leak detection. In Fig. 9, the four modules, Andersen's analysis, Intra-Refinement, Wrapper Detector and BDD, are from Open64. All the remaining modules are developed by us.

Unlike FASTCHECK, which reasons about paths using a SAT solver, SABER encodes paths using BDDs. There are some advantages for doing so. First, the number of BDD variables used (for encoding branch conditions) is kept to a minimum. Second, it plays up the strengths of BDDs by exposing opportunities for path redundancy elimination. Third, the paths combined at a join point are effectively simplified (e.g., with $C_1 \vee \neg C_1$ being reduced into true).

Following [3], a test comparing the allocated value against NULL is replaced with an appropriate truth value. For example, if `p = malloc()` is analyzed, `p == null` is replaced by `false` since the analysis considers only the cases where the allocation is successful. This simplification is generalized to tests of the form `q == e`, where e is an expression [3].

To guarantee efficiency without losing much accuracy, the size of a backward slice is bounded by 100 nodes. As illustrated in Fig. 12 and discussed in Section 6.2, the backward slices in a program are mostly small, with no more than 10 SVFG nodes. A source is ignored if the limit is exceeded by its backward slice. No new leaks are found in the programs evaluated after the limit has been increased to 500.

6 EXPERIMENTAL EVALUATION

We evaluate SABER using the 15 SPEC2000 C programs (620 KLOC) and seven open-source applications (1.7 MLOC), listed in Table 4. We compare SABER with ATHENA [13] (user specification), CONTRADICTION [22] (data-flow analysis), SPARROW [11] (abstract interpretation), CLANG, which stands here for its static analyzer (version checker-259) [10] (symbolic execution) and FASTCHECK [3] (sparse value-flow analysis). Of these tools, only FASTCHECK and CLANG are publicly available. By using SPEC2000, a comparison between SABER and these tools is made possible based on the data available in their papers.

Given a program, the analysis time of SABER is reported as the average time over three runs. Similarly, the compile time of Open64 is also calculated this way.

When assessing SABER, we consider three criteria: (1) *practicality* (its competitiveness against other detectors), (2) *efficiency* (its analysis time) and (3) *accuracy* (its ability to detect memory leaks with a low false positive rate). Our results presented and analyzed below show that SABER has achieved its design objectives outlined earlier. All our experiments were done on a platform consisting of a 3.0 GHZ Intel Core2 Duo processor with 16 GB memory, running RedHat Enterprise Linux 5 (kernel version 2.6.18).

TABLE 4
Program Characteristics

Program	Characteristics				
	#Functions	#Pointers	#Loads /Stores	#Allocation Sites	#Free Sites
ammp	182	9829	1636	37	30
art	29	600	103	11	1
bzip2	77	1672	434	10	4
crafty	112	11883	3307	12	16
equake	30	1203	408	29	0
gap	857	61435	16841	2	1
gcc	2256	134380	51543	161	19
gzip	113	3004	586	3	3
mcf	29	1317	526	4	3
mesa	1109	44582	17302	82	76
parser	327	8228	2597	1	0
perlbmk	1079	54816	16885	148	2
twolf	194	20773	8657	185	1
vortex	926	40260	11256	9	3
vpr	275	7930	2160	130	68
bash	2700	17830	6855	112	58
cluster	96	3214	1241	124	178
droplet	468	15111	4129	251	193
httpd	3000	60027	18450	21	18
icecast	603	15098	9779	235	235
sendmail	2656	107242	22191	296	136
wine	77829	1330840	137409	571	231

6.1 Practicality

Tables 1 and 2 compare SABER with several other static leak detectors using some or all of the 15 SPEC2000 C programs. In Table 2, the total size of the SPEC2000 C programs analyzed by each tool is given. Note that the same benchmark may have different code sizes if its different versions are used. The data for CLANG and SABER are produced in this work and the data for the others are obtained from their cited papers. As remarked earlier in Section 1, speed numbers should be regarded as rough estimates.

SABER reports 85 faults among 105 leak warnings while SPARROW reports 81 among 96 warnings (without being able to compile `perlbmk` [11]). FASTCHECK detects 59 faults among 67 warnings. ATHENA finds about the same number of leaks as FASTCHECK with a lower false positive rate but is much slower. Both CONTRADICTION and CLANG find much fewer leaks. SABER detects consistently more faults than the others while keeping its false positive rate at about 19 percent for SPEC2000. In addition, SABER achieves this level of accuracy by running a few times slower than FASTCHECK but is a lot faster than the other tools.

To compare SABER further with FASTCHECK and CLANG, which are open-source tools, we have manually checked all the leak warnings issued. In the case of FASTCHECK, one of its authors graciously provided us their fault report. SABER succeeds in finding a superset of the faults reported by each. SABER always detects no fewer faults than FASTCHECK because SABER's value-flow graph is more precise. SABER performs better than CLANG because CLANG is intraprocedural. During our experiments, its "experimental.unix.Malloc" checker is

TABLE 5
SABER's Fault Counts and Analysis Times

Program	Size (KLOC)	Time (secs)	Fault Count	#False Alarms
ammp	13.4	0.55	20	0
art	1.2	0.01	1	0
bzip2	4.7	0.04	1	0
crafty	21.2	0.83	0	0
equake	1.5	0.04	0	0
gap	71.5	4.00	0	0
gcc	230.4	21.21	42	5
gzip	8.6	0.08	1	0
mcf	2.5	0.03	0	0
mesa	61.3	10.10	7	4
parser	11.4	0.28	0	0
perlbmk	87.1	18.52	8	4
twolf	20.5	2.12	5	0
vortex	67.3	2.90	0	4
vpr	17.8	0.31	0	3
bash	100	22.03	8	2
cluster	10.7	1.81	12	4
droplet	33.2	29.70	9	3
httpd	128.1	10.65	0	0
icecast	22.3	5.54	13	5
sendmail	115.20	32.97	5	0
wine	1338.1	421.60	122	23
total	2368.0	585.32	254	57

used to enable leak detection. By analyzing a function individually without considering its callers and callees, the information from outside (via its parameters and returns at callsites) is assumed to be unknown or symbolic. Thus, any objects created inside callees cannot be analyzed, thereby causing CLANG to miss many faults.

In addition to SPEC2000, we have also evaluated SABER using seven open-source C applications, totalling 1.7 MLOC: `wine-0.9.24` (a tool that allows windows applications to run on Linux), `icecast-2.3.1` (a streaming media server), `bash-3.1` (a UNIX shell), `cluster-3.0` (clustering algorithms), `droplet-3.0` (a cloud storage client library), `httpd-2.0.64` (an Apache HTTP server) and `sendmail-8.14.2` (an internet email server).

Table V summarizes the accuracy and analysis times for the 15 SPEC2000 C programs and seven applications. In `wine`, the largest in our suite, SABER finds 122 faults with 23 false positives in about 421.6 secs, i.e., which is roughly the amount of time taken in compiling `wine` under "-O2". Overall, SABER finds 254 leaks at a false positive rate of 18.3 percent. To the best of our knowledge, SABER is the fastest leak detector scalable to millions of lines of code at this accuracy.

6.2 Efficiency

SABER is fully implemented in the Open64 compiler. We investigate and analyze its efficiency further by comparing the analysis times used by SABER with the compile times consumed by Open64 under "-O2" for our test suite. As shown in Fig. 10, both are similar across all the programs, indicating that SABER is promising to be incorporated into an industry-strength compiler for static leak detection.

For some small and medium programs such as `art`, `ammp`, `equake`, `gzip`, `mcf`, `parser` and `vpr`, SABER's analysis times are significantly less than Open64's compile times. For some large programs like `droplet`, `mesa`, `perlbmk` `sendmail` and `wine`. SABER's analysis times are slightly longer since these programs each have a relatively large number of abstract heap objects to be analyzed as shown in Table 4. For `gcc`, the largest in SPEC2000, SABER can analyze it faster than Open64 compiles it. This is the case because the most of the backward slices considered during all-path analysis are small. The effectiveness of solving our sink reachability problem in backward rather than forward slices is illustrated in Fig. 12.

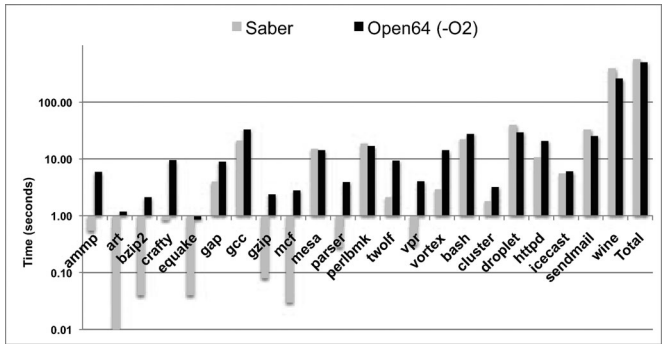

Fig. 10. Comparing SABER's analysis times and Open64's compile times (under "-O2").

SABER analyzes a program by going through its four phases in Fig. 9. To understand their relative costs, Table 4 gives some basic characteristics about the programs being analyzed. For each program, Columns 2-6 give the number of functions, pointers, loads/stores, abstract objects (i.e., allocation sites) and free sites in the program. The presence of these many loads/stores indicates the importance of tracking the values flowing into address-taken pointers in this work. Fig. 12 analyzes the leak detection phase further, by comparing the percentages of nodes in SVFG and functions present in the forward slices $\mathcal{F}_{src}$ built during some-path analysis and those in the backward slices $\mathcal{B}_{src}$ built during all-path analysis (Section 4.4). Recall that it is on $\mathcal{B}_{src}$ that SABER reasons about sink reachability on all paths. Most of the backward slices are smaller, not exceeding 10 SVFG nodes.

From Fig. 11, we can examine SABER's analysis times distributed among its four phases in our test suite. In total, their percentage distributions are pre-analysis (30.1 percent), full-sparse SSA (14.3 percent), SVFG (34.9 percent) and leak detection (20.7 percent). The pre-analysis and SVFG phases together dominate the analysis times for all the programs. The SSA phase seems to take some noticeable

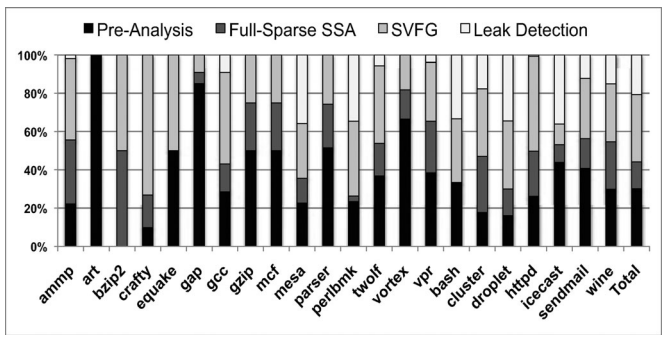

Fig. 11. Percentage distribution of SABER's analysis times among its four phases.

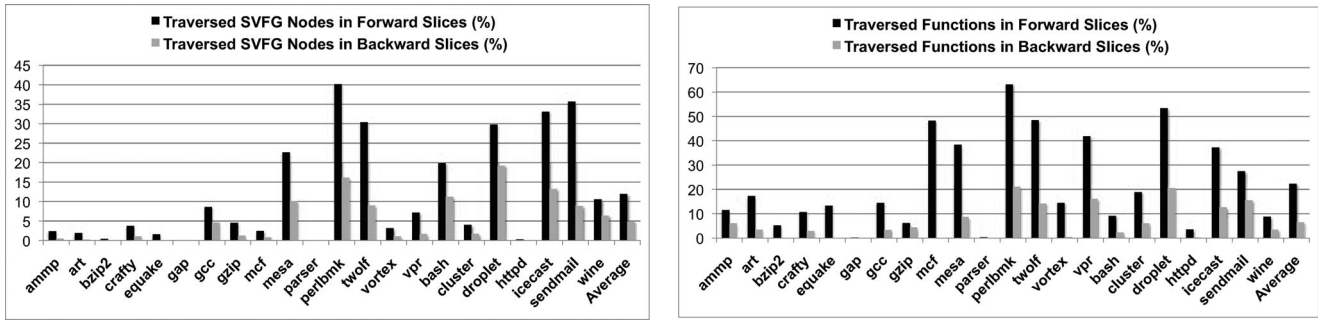

Fig. 12. Traversed functions (%) and SVFG nodes (%) in forward $\mathcal{F}_{src}$ and backward $\mathcal{B}_{src}$ slices.

fractions of the total times in some large programs, such as `gcc`, `httpd` and `wine`, which have relatively a large number of loads/stores (and callsites). In the case of `ammp`, `bzip2`, `gzip`, `mcf`, `vpr` and `parser`, few pointers and allocation sites are found (Table 4) and their analysis times are all within 1 sec (Table 5). So the percentage distributions should be interpreted in this context.

Finally, the leak detection phase is relatively fast since, as shown in Table 4, the portions of the SVFGs being considered during all-path analysis are small. On average, only 6.48 percent of the functions and 5.08 percent of the nodes in the SVFGs are traversed. In the case of `gap`, `parser` and `httpd`, which are medium programs with many pointers, little times are consumed in this phase. A glance at Table 4 reveals that these programs each have few abstract heap objects to be checked. In `gap`, most of its computations are done on global data structures. In `parser` and `httpd`, a large pool of memory is allocated at the beginning and used frequently after. In `twolf` and `vpr`, there are many allocation sites but most of the objects created reach *GLOBAL* as discovered during some-path analysis. In contrast, `mesa`, `perlbmk`, `bash`, `droplet` and `icecast` stay longer than the other programs in the leak detection phase, because their backward slices $\mathcal{B}_{src}$ are relatively large (Table 4). Some programs such as `gcc`, `sendmail` and `wine` have many allocation sites but are relatively fast to analyze in this phase. This is because many of their objects are either never freed or reach *GLOBAL* during some-path analysis. As for `wine`, SABER starts with 571 abstract objects. After the some-path analysis is done, there are 105 that are never freed and 270 that reach *GLOBAL*. So only 196 objects need to be further checked relatively more costly during all-path analysis.

6.3 Accuracy

We examine the causes for some interesting leaks reported by SABER to analyze further its accuracy. In the FASTCHECK paper [3], the 15 SPEC2000 C programs and `bash` are also considered. When comparing SABER with FASTCHECK, we refer to the fault report on these programs communicated to us by one of its authors. We also examine six representative scenarios with some leaks detected by SABER but missed by FASTCHECK to highlight the importance of tracking value flows into address-taken pointers.

As shown in Table 5, SABER finds 254 faults with 57 false positives, achieving a false positive rate of 18.3 percent. Let us consider SPEC2000 first. All the faults (20) in `ammp` are conditional leaks, caused when functions do not free memory when returning on errors. All these faults can also be found by FASTCHECK as they require only value-flow analysis for top-level pointers. All faults reported in `gcc` are due to mis-handling of strings. Most of these (with three inside loops) are related to calls to `concat`. For `mesa`, all the seven faults found by SABER but missed by FASTCHECK require value-flow analysis for address-taken pointers. Some conditional leaks at a switch statement and some never-freed leaks are discussed below. Among the eight faults reported for `perlbmk` by SABER, only two are also reported by FASTCHECK. The remaining six involve heap objects being passed into a field of a local struct variable or passed outside from a callee via dereferenced formal parameters.

Let us move to the seven applications, of which only `bash` is also analyzed by FASTCHECK [3]. In `bash`, FASTCHECK finds two of the eight faults found by SABER. For the other six faults, two share a similar pattern. A function allocates two objects, one to a base pointer p and one to $*p$. If the second allocation for $*p$ fails, it returns without freeing the object allocated for p. In the case of `icecast`, with 12 faults reported, three are related to mis-handling of strings and the other nine (with some analyzed below) all happen when a function does not free objects that are allocated but unsuccessfully inserted into a list. SABER detects two conditional leaks in `sendmail` at switch statements. SABER finds 122 faults in `wine`, with 105 never freed and the remaining ones as conditional leaks (caused for a variety of reasons). A scenario similar to our motivating example is discussed below.

Below we examine six representative scenarios `mesa` in SPEC2000 and `cluster`, `droplet`, `icecast` and `wine` in open-source code. There are a total of nine leaks: seven require value-flow analysis for address-taken pointers and two can be found by analyzing top-level pointers only.

Consider the code region from `mesa` given in Fig. 13a. At lines 362 and 385, two heap objects are allocated and assigned to `textImage` and its field `Data`, respectively. However, both objects are conditional leaks when the format of `testImage` created does not match any that is listed at the switch statement. In this case, the function returns directly at line 478, without freeing the two heap objects allocated earlier.

Consider the code fraction for initializing some arrays in `cluster` shown in Figs. 13b and 13c. In Fig. 13b, `u` and `v` are used as two-dimensional arrays and `w` and `m` as one-dimensional arrays. However, when the condition at line 1,187 holds, the one-dimensional arrays created at lines 1,170 and 1,179 for initializing `u` and `v`, respectively, are not

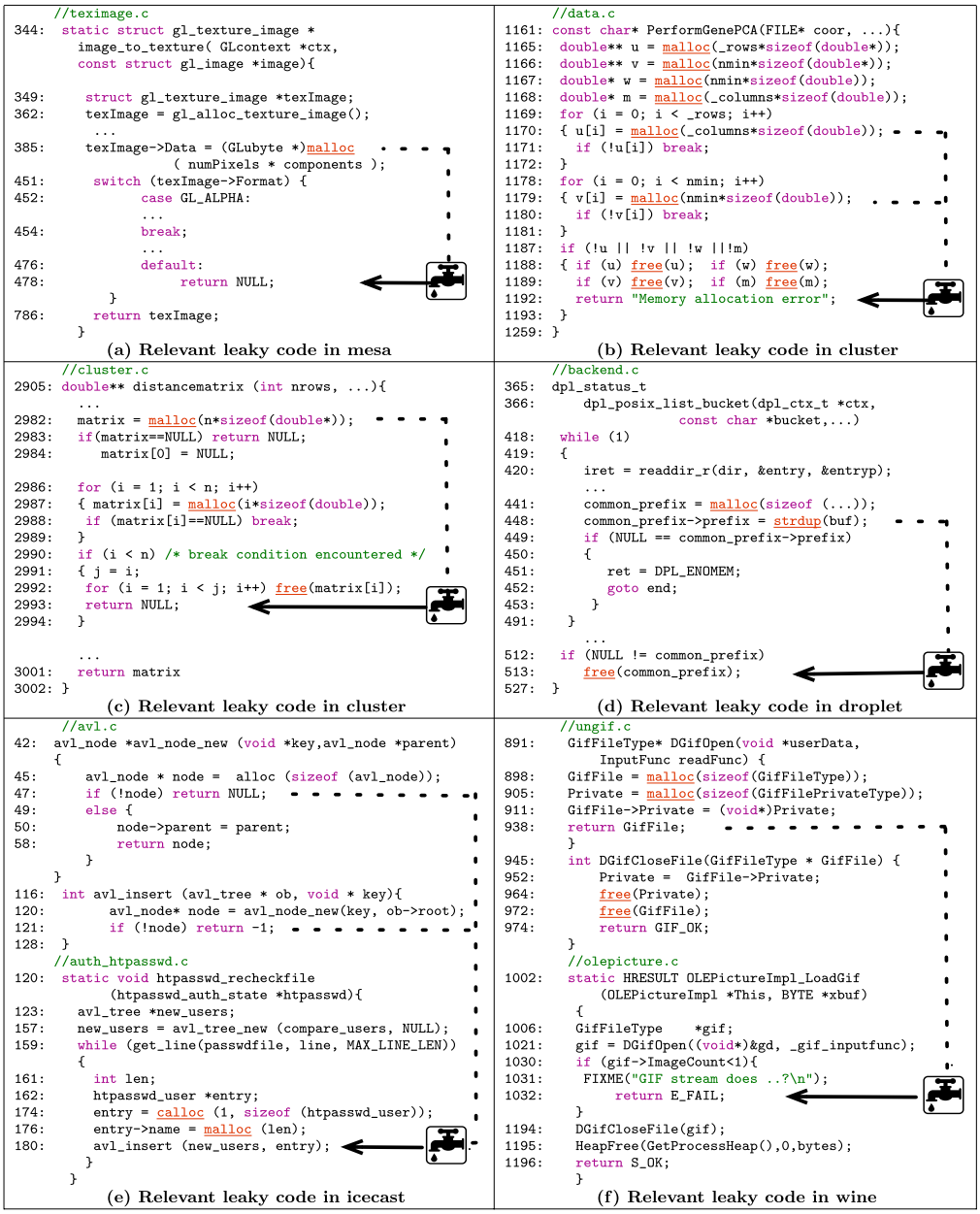

Fig. 13. Six scenarios with conditional leaks requiring value-flow analysis for address-taken variables.

freed. A similar case happens in Fig. 13c, except that the one-dimensional arrays pointed by the individual elements of the two-dimensional `matrix` array are freed but `matrix` itself leaks at line 2993.

Consider the code region from `droplet` given in Fig. 13d. Two heap objects are allocated and assigned to `common_prefix` and its field `prefix` at lines 441 and 448. Only the object pointed by `common_prefix` is freed but the other is not.

Let us look at the two leaks in `icecast` as shown in Fig. 13e. At lines 174 and 176, `entry` and its field `name` are assigned a heap object each. Subsequently at line 180, `avl_insert` is called to insert `entry` into the `new_users` list. However, the insertion fails when the test at line 121 in `avl_insert` succeeds. Then the two objects leak. There are nine occurrences of this leak pattern in `icecast`.

Finally, we discuss the two leaks in `wine` in Fig. 13f, which are similar to the two illustrated in our motivating example. In function `OLEPictureImpl_LoadGif`, `GifOpen` is called at line 1,021 so that two heap objects are allocated at lines 898 and 905: one is passed to `gif` and the other to the field `private` of `GifFile`. At the end of `OLEPictureImpl_LoadGif`, there is a call to `DGifCloseFile` to free the two objects. However, the two objects are never freed when the test at line 1030 sitting between the two calls evaluates to true.

6.4 Limitations

Like nearly all static memory leak detectors, SABER is neither complete (by issuing false positives) nor sound (by missing faults).

The false positives happened for several different reasons: not recognizing infeasible paths (in `mesa`, `bash` and `wine`), treating multi-dimensional arrays monolithically (in `vpr`), bounding the number of loop iterations

(in `vortex`, `icecast` and `wine`) and approximating aliases conservatively in terms of memory regions (in `perlbmk` and `wine`).

Similarly, the false negatives also happened for several different reasons. Static detectors analyze usually the first N iterations of a loop or recursion cycle. To compare SABER with FASTCHECK, $N = 1$ is also used. Thus, SABER shares the same limitations as FASTCHECK in these aspects. When $N = 2$, SABER has managed to detect one more leak (in function `symbol_search` in `wine`). In the worst case, all the iterations of a loop or recursion cycle must be analyzed to avoid missing any leaks. This is the case, for example, when one loop is used to initialize different elements of an array with different heap objects but another loop is used to free all the objects but the last.

In addition, both SABER and FASTCHECK, like many other tools, are not sound in handling pointer arithmetic by treating, for example, an occurrence of $x + e$ as an occurrence of x. Like other tools, SABER may also miss faults due to imprecision in modelling the heap. For example, an abstract heap object that represents two different concrete objects along two different call paths is considered not to leak if the abstract object is freed along one of the two paths.

SABER handles global variables similarly as in FASTCHECK and SPARROW. This is not sound since the leaks reachable by *GLOBAL* are not tracked.

Finally, SABER ignores an allocation source if its backward slice exceeds a given threshold. This may potentially cause some leaks to go undetected.

7 Related Work

There are a number of reported attempts on memory leak detection using static analysis [3], [9], [10], [11], [13], [22], [30], [33] or dynamic analysis [1], [5], [20], [35], [36]. The SABER approach can speed up existing static techniques by using a full-sparse representation to track the flow of values through assignments.

Static Memory Leak Detection. There has been a lot of research devoted to checking memory leaks statically [3], [9], [11], [15], [22], [32]. SATURN [33] reduces the problem of memory leak detection to a boolean satisfiability problem and then uses a SAT solver to identify errors. Its analysis is context-sensitive and intraprocedurally path-sensitive. So SATURN can find some leaks missed by SABER. By solving essentially a constraint formulation of a data-flow analysis problem, however, SATURN scales to around 50 LOC/sec (on a 2.8 GHz Intel dual Xeon with 4 GB memory) when analyzing some common programs [33]. BDDs are also used previously to represent and reason about program paths [31], [34]. ATHENA [13] finds path-sensitive faults guided by user specifications, without handling some language features such as function pointers. SABER resolves function pointers during its pre-analysis. CLANG [10] is a source-code analysis tool that can find memory leaks in C and Objective-C programs based on symbolic execution. Being intraprocedural, it assumes unknown or symbolic values for the formal parameters of a function and the returned values from its callsites. SPARROW [11] relies on abstract interpretation to detect leaks in C programs. It models a function using a parameterized summary and uses the summary to analyze all the call sites to the function. FASTCHECK [3] detects memory leaks by using a semi-sparse representation to track the flow of values through top-level pointers only. It is fast but limited to analyzing allocation sites whose values flow into top-level pointers only. CONTRADICTION [22] performs a backward data-flow analysis to disprove the presence of memory leaks. CLOUSEAU [9] is a flow- and context-sensitive memory leak detector, based on an ownership model. Compared to the other tools, CONTRADICTION and CLOUSEAU issue relatively more false positives. SABER as presented in this paper is the first static tool for detecting memory leaks using a full-sparse value-flow graph.

Dynamic Memory Leak detection. Such tools [1], [5], [18] detect leaks by instrumenting and running a program based on test inputs. However, dynamic detectors tend to miss faults although their false positive rates can be kept low. For example, we ran valgrind [18] on the same 15 SPEC2000 C programs using the reference inputs provided. More than 90 percent leaks reported by SABER cannot be detected.

Sparse Pointer Analysis. Unlike iterative data-flow pointer analyses, their recent sparse incarnations [7], [8], [14], [31], [38] avoid propagating information unnecessarily guided by pre-computed def-use chains. Earlier, Hardekopf and Lin presented a semi-sparse flow-sensitive analysis [8]. By putting top-level pointers in SSA, their def-use chains can be exposed directly. Recently, they have generalized their work by making it full-sparse [7]. This is done by using Andersen-style flow-insensitive pointer analysis to compute the required def-use information in order to build SSA for all variables. Yu et al. [38] introduced LevPA, a flow- and context-sensitive pointer analysis on full-sparse SSA. Pointer resolution and SSA construction are performed together, level by level, in decreasing order of their points-to levels. All pointer analyses require a heap cloning method to partition the infinite-sized heap into a finite a number of abstract objects [12], [21], [29].

Interprocedural Side-Effect Analysis. There are techniques for computing interprocedural modification side-effects for FORTRAN [2], C [24] and Java with a closed-world assumption [17], [25] and in the presence of dynamic class loading [19], [37]. SABER uses a Mod-Ref analysis that restricts itself to the modification and reference side-effects on nonlocal memory locations (due to Assumption 1).

8 Conclusion

Memory leaks are common errors affecting many programs including OS kernels, desktop applications and web services. In this paper, we have introduced SABER, a static detector for finding memory leaks in C programs. By using a full-sparse value-flow graph to track the flow of values from allocation to free sites through both top-level and address-taken pointers, SABER is effective at finding leaks in SPEC2000 and seven open-source applications, by detecting a total of 254 leaks at a false positive rate of 18.3 percent.

Acknowledgments

The authors would like to thank the reviewers for their valuable comments. This work was supported by the

Australian Research Council (ARC) grants, DP110104628 and DP130101970, and a grant from Oracle Labs.

REFERENCES

[1] D. Bruening and Q. Zhao, "Practical Memory Checking with Dr. Memory," *Proc. IEEE/ACM Ninth Ann. Int'l Symp. Code Generation and Optimization (CGO '11)*, 2011.

[2] D. Callahan and K. Kennedy, "Analysis of Interprocedural Side Effects in a Parallel Programming Environment," *J. Parallel and Distributed Computing*, vol. 5, no. 5, pp. 517-550, Oct. 1988.

[3] S. Cherem, L. Princehouse, and R. Rugina, "Practical Memory Leak Detection using Guarded Value-Flow Analysis," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '07)*, 2007.

[4] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich, "Effective Representation of Aliases and Indirect Memory Operations in SSA Form," *Proc. Sixth Int'l Conf. Compiler Construction (CC '96)*, 1996.

[5] J. Clause and A. Orso, "LEAKPOINT: Pinpointing the Causes of Memory Leaks," *Proc. ACM/IEEE 32nd Int'l Conf. Software Engineering (ICSE '10)*, 2010.

[6] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. Programming Languages and Systems*, vol. 13, no. 4, pp. 451-490, Oct. 1991.

[7] B. Hardekopf and C. Lin, "Flow-Sensitive Pointer Analysis for Millions of Lines of Code," *Proc. IEEE/ACM Ninth Ann. Int'l Symp. Code Generation and Optimization (CGO '11)*, 2011.

[8] B. Hardekopf and C. Lin, "Semi-Sparse Flow-Sensitive Pointer Analysis," *Proc. 36th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '09)*, 2009.

[9] D.L. Heine and M.S. Lam, "A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '03)*, 2003.

[10] http://clang-analyzer.llvm.org/, 2014.

[11] Y. Jung and K. Yi, "Practical Memory Leak Detector Based on Parameterized Procedural Summaries," *Proc. Seventh Int'l Symp. Memory Management (ISMM '08)*, 2008.

[12] C. Lattner, A. Lenharth, and V. Adve, "Making Context-Sensitive Points-To Analysis with Heap Cloning Practical for the Real World," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '07)*, pp. 278-289, 2007.

[13] W. Le and M.L. Soffa, "Generating Analyses for Detecting Faults in Path Segments," *Proc. Int'l Symp. Software Testing and Analysis (ISSTA '11)*, 2011.

[14] L. Li, C. Cifuentes, and N. Keynes, "Boosting the Performance of Flow-Sensitive Points-To Analysis Using Value Flow," *Proc. 19th ACM SIGSOFT Symp. and 13th European Conf. Foundations of Software Eng. (FSE '11)*, 2011.

[15] V.B. Livshits and M.S. Lam, "Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs," *Proc. Ninth European Software Eng. Conference held jointly with 11th ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (FSE '03)*, 2003.

[16] Y. Lu, L. Shang, X. Xie, and J. Xue, "An Incremental Points-to Analysis with CFL-Reachability," *Proc. 22nd Int'l Conf. Compiler Construction (CC '13)*, pp. 61-81, 2013.

[17] R. Madhavan, G. Ramalingam, and K. Vaswani, "Purity Analysis: An Abstract Interpretation Formulation," *Proc. 18th Int'l Conf. Static Analysis (SAS '11)*, 2011.

[18] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '07)*, 2007.

[19] P.H. Nguyen and J. Xue, "Interprocedural Side-Effect Analysis and Optimisation in the Presence of Dynamic Class Loading," *Proc. 28th Australasian Conf. Computer Science (ACSC '05)*, pp. 9-18, 2005.

[20] G. Novark, E. Berger, and B. Zorn, "Efficiently and Precisely Locating Memory Leaks and Bloat," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '09)*, 2009.

[21] E. Nystrom, H. Kim, and W. Hwu, "Importance of Heap Specialization in Pointer Analysis," *Proc. Fifth ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Eng. (PASTE '04)*, pp. 43-48, 2004.

[22] M. Orlovich and R. Rugina, "Memory Leak Analysis by Contradiction," *Proc. 13th Int'l Conf. Static Analysis (SAS '06)*, 2006.

[23] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," *Proc. 22nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '95)*, 1995.

[24] B.G. Ryder, W.A. Landi, P.A. Stocks, S. Zhang, and R. Altucher, "A Schema for Interprocedural Modification Side-Effect Analysis with Pointer Aliasing," *ACM Trans. Programming Languages and Systems*, vol. 23, no. 2, pp. 105-186, Mar. 2001.

[25] A. Salcianu and M.C. Rinard, "Purity and Side Effect Analysis for Java Programs," *Proc. Sixth Int'l Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI '05)*, pp. 199-215, 2005.

[26] L. Shang, Y. Lu, and J. Xue, "Fast and Precise Points-to Analysis with Incremental CFL-Reachability Summarisation: Preliminary Experience," *Proc. IEEE/ACM 27th Int'l Conf. Automated Software Eng. (ASE '12)*, pp. 270-273, 2012.

[27] L. Shang, X. Xie, and J. Xue, "On-Demand Dynamic Summary-Based Points-to Analysis," *Proc. 10th Int'l Symp. Code Generation and Optimization (CGO '12)*, 2012.

[28] M. Sridharan and R. Bodík, "Refinement-Based Context-Sensitive Points-To Analysis for Java," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '06)*, 2006.

[29] Y. Sui, Y. Li, and J. Xue, "Query-Directed Adaptive Heap Cloning for Optimizing Compilers," *Proc. IEEE/ACM Int'l Symp. Code Generation and Optimization (CGO '13)*, pp. 1-11, 2013.

[30] Y. Sui, D. Ye, and J. Xue, "Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis," *Proc. Int'l Symp. Software Testing and Analysis (ISSTA '12)*, 2012.

[31] Y. Sui, S. Ye, J. Xue, and P. Yew, "SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA," *Proc. Ninth Asian Conf. Programming Languages and Systems (APLAS '11)*, 2011.

[32] E. Torlak and S. Chandra, "Effective Interprocedural Resource Leak Detection," *Proc. ACM/IEEE 32nd Int'l Conf. Software Eng. (ICSE '10)*, 2010.

[33] Y. Xie and A. Aiken, "Context-and Path-Sensitive Memory Leak Detection," *Proc. 10th European Software Eng. Conference held jointly with 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (FSE '05)*, 2005.

[34] Y. Xie and A. Aiken, "Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability," *ACM Trans. Programming Languages and Systems*, vol. 29, article 26, 2007.

[35] G. Xu, M. Bond, F. Qin, and A. Rountev, "LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks," *Proc. 32nd ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '11)*, 2011.

[36] G. Xu and A. Rountev, "Precise Memory Leak Detection for Java Software Using Container Profiling," *Proc. 30th Int'l Conf. Software Eng. (ICSE '08)*, 2008.

[37] J. Xue, P.H. Nguyen, and J. Potter, "Interprocedural Side-Effect Analysis for Incomplete Object-Oriented Software Modules," *J. Systems and Software*, vol. 80, no. 1, pp. 92-105, 2007.

[38] 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," *Proc. IEEE/ACM Eighth Ann. Int'l Symp. Code Generation and Optimization (CGO '10)*, 2010.

Yulei Sui received the bachelor's and master's degrees in computer science from Northwestern Polytechnical University, Xi'an, China, in 2008 and 2011. He has been working toward the PhD degree in programming languages and compilers group at the University of New South Wales since 2010. He is broadly interested in the research field of software engineering and programming languages, particularly interested in static and dynamic program analysis for software bug detection and compiler optimizations. He was a research intern in the Program Analysis Group for Memory Safe C project in Oracle Lab Australia in 2013. He was an Australian IPRS scholarship holder and a Best Paper Award winner at CGO '13.

Ding Ye received the bachelor's degree in information security from the Huazhong University of Science and Technology, Wuhan, China, in 2008, and the master's degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2011. He has been working toward the PhD degree under the supervision of Professor Jingling Xue at the University of New South Wales, Australia, since 2011. His research interests include program analysis and compiler techniques.

Jingling Xue received the BSc and MSc degrees in computer science and engineering from Tsinghua University in 1984 and 1987, respectively, and the PhD degree in computer science and engineering from Edinburgh University in 1992. He is currently a professor in the School of Computer Science and Engineering, University of New South Wales, Australia, where he heads the Programming Languages and Compilers Group. His main research interest has been programming languages and compilers for about 20 years. He is currently supervising a group of postdocs and PhD students on a number of topics including programming and compiler techniques for multi-core processors and embedded systems, concurrent programming models, and program analysis for detecting bugs and security vulnerabilities. He is currently an associate editor of the *IEEE Transactions on Computers, Software: Practice and Engineering, International Journal of Parallel, Emergent and Distributed Systems*, and *Journal of Computer Science and Technology*. He has served in various capacities on the Program Committees of many conferences in his field.