Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis

Yulei Sui Ding Ye Jingling Xue

Programming Languages and Compilers Group
School of Computer Science and Engineering, UNSW, Australia

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 leak detection. 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 leaks in a program by solving a graph reachability problem on its SVFG.

SABER, which is fully implemented in Open64, is effective at detecting 211 leaks in the 15 SPEC2000 C programs and five applications, while keeping the false positive rate at 18.5%. We have also compared SABER with FASTCHECK (which analyzes allocated objects flowing only into top-level pointers) and SPARROW (which handles all allocated objects using abstract interpretation) using the 15 SPEC2000 C programs. SABER is as accurate as SPARROW but is 14.2X faster and reports 40.7% more bugs than FASTCHECK at a slightly higher false positive rate but is only 3.7X slower.

Categories and Subject Descriptors

D 2.4 [**Software/Program Verification**]: Reliability; D 3.4 [**Processors**]: Compilers, Memory Management; F 3.2 [**Semantics of Programming Languages**]: Program Analysis

General Terms

Algorithms, Languages, Verification

Keywords

Memory Leaks, Static Analysis, Sparse Value-Flow Analysis

1. INTRODUCTION

This paper introduces a new static detector, SABER, which is fully implemented in the Open64 compiler, for detecting memory leaks in C programs. Table 1 compares SABER with several other static detectors based on published and self-produced data on their scalability and accuracy in analysing the 15 SPEC2000 C programs (totalling 620 KLOC). These results, together with those reported later on analysing five applications (totalling 1.7 MLOC), show that SABER has met its design objectives, as discussed below.

Table 1: Comparing SABER with other static detectors on analysing the 15 SPEC2000 C programs. The data for CLANG, which stands here for its static analysis tool, and SABER are from this paper while the data for the other three tools are from the papers cited. Saturn [20], which has no published data on SPEC2000, runs at 50 LOC/sec with a false positive rate of 10% [2].

Leak Detector	Speed (LOC/sec)	Bug Count	False Positive Rate (%)
CONTRADICTION [14]	300	26	56
CLANG [8]	400	27	25
SPARROW [9]	720	81	16
FASTCHECK [2]	37,900	59	14
SABER	**10,220**	**83**	**19**

1.1 Motivations and 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 CONTRADICTION [14] (data-flow analysis), SATURN [20] (Boolean satisfiability), SPARROW [9] (abstract interpretation), CLANG [8] (symbolic execution) and FASTCHECK [2] (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 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 analysing 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 Table 1, 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 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 [2]. 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 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. At this stage in its development, 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 [5, 6, 10, 18, 24]. Instead of resolving pointers iteratively in a data-flow analysis framework, sparse pointer analysis interleaves pointer resolution and def-use or SSA construction to obtain pointer information more quickly.

1.2 Challenges

As shown in Table 1, Sparrow is more accurate than Fastcheck but at a 52.6X slowdown. 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

We have designed and implemented Saber in the Open64 compiler using a full-sparse value-flow analysis for all memory locations (top-level and address-taken pointers). Saber operates in four phases, by (1) performing a pre-analysis (to discover pointer/aliasing information), (2) putting all locations in SSA form, (3) building a sparse representation of value flows, called a sparse value-flow graph (SVFG), that captures def-use chains and value flows via assignments for all locations, and (4) detecting leaks in a program via solving a graph (sink) reachability problem on its SVFG. 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.

This paper makes the following 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 showing 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 bug detection tools.
- Saber is effective at detecting 211 leaks in the 15 SPEC2000 C programs and five other open-source applications, while keeping the false positive rate at 18.5%.
- Saber is as accurate as Sparrow but is 14.2X faster and reports 40.7% more bugs than Fastcheck at a slightly higher false positive rate but is only 3.7X slower, measured by the 15 SPEC2000 C programs (Table 1).

2. THE SABER DETECTOR

Saber detects memory leaks using a full-sparse value-flow analysis. This requires all memory locations, represented by both top-level variables and address-taken variables, to be put in SSA form. In SSA form, each variable 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.

The conversion to SSA implies that the def-use chains for both top-level and address-taken variables need to be determined. For top-level variables, this is done trivially just like for scalars. For address-taken variables, a pointer analysis is required due to the existence of indirect defs and uses through pointers. For improved precision, some degree of flow-sensitivity is usually considered. Then there are two approaches to determining the def-use chains for address-taken variables. 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 [5, 6, 18, 24] 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.

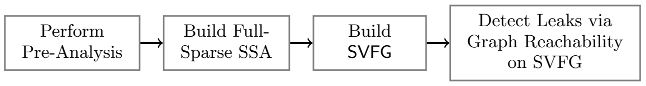
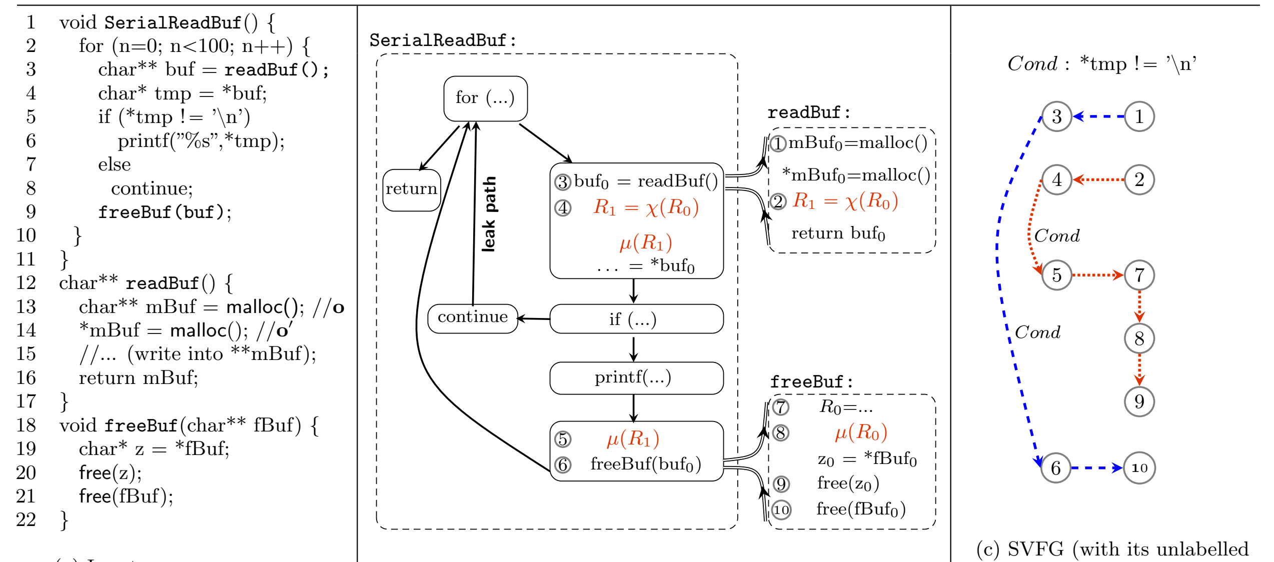


Figure 1: Structure of the Saber detector.

Saber proceeds in four phases, as shown in Figure 1. Their functionalities are described below, with details given in Sections 2.3 – 2.6. 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.



```
1   void SerialReadBuf() {
2     for (n=0; n<100; n++) {
3       char** buf = readBuf();
4       char* tmp = *buf;
5       if (*tmp != '\n')
6         printf("%s",*tmp);
7       else
8         continue;
9       freeBuf(buf);
10    }
11  }
12  char** readBuf() {
13    char** mBuf = malloc(); //o
14    *mBuf = malloc(); //o′
15    //... (write into **mBuf);
16    return mBuf;
17  }
18  void freeBuf(char** fBuf) {
19    char* z = *fBuf;
20    free(z);
21    free(fBuf);
22  }
```

(a) Input program (b) Full-sparse SSA (c) SVFG (with its unlabelled edges being guarded by `true`)

Figure 2: A motivating example

Phase 2: Full-Sparse SSA This is built for each function individually, by considering all memory locations. We adopt a balanced model to represent memory 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: 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 (SVFG), 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 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 free sites (sinks).

2.1 Program Representation

In the canonical form, a statement in the CFG of an input program is one of the following: (1) an assignment of the form, $x = \&y$ (address), $x = y$ (copy), $*x = y$ (store) or $x = *y$ (load), where x and y are local or global variables, (2) a call $b = f(a_1, \dots, a_n)$, where b, $a_1, \dots, a_1$ are all local variables, (3) a return statement, return r, where r is a local variable, and (4) a two-way branch (i.e., an if-statement).

During the conversion to SSA, three new types of statements are introduced: ϕ, μ, and χ. The ϕ functions are added at join points as is standard. Following [3], indirect defs and uses at loads and stores are represented by using μ and χ functions. Each load $x = *y$ in the input program is annotated with a function $\mu(v)$ for each variable v that may be read by the load. Similarly, each store $*x = y$ in the input program is annotated with a function $v = \chi(v)$ for each variable v that may be defined by the store. When converted to SSA form, each $\mu(v)$ is treated as a use of v and each $v = \chi(v)$ is treated as both a def and use of v.

To understand this asymmetric treatment of μ and χ, suppose $v = \chi(v)$ (associated with $*x = y$) becomes $v_m = \chi(v_n)$ after SSA conversion. If x uniquely points to v, which represents a concrete memory location, then v_m can be strongly updated. In this case, v_m receives whatever y points to and the information in v_n is ignored. Otherwise, v_m must incorporate the pointer information from both v_n and y.

2.2 A Motivating Example

Let us use an example in Figure 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` as depicted in Figure 7(d). In Figure 2(a), **readBuf** is called in a for loop in **SerielReadBuf**. Every time when **readBuf** is called, a single-char buffer formed by two objects 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.

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

Phase 1: Pre-Analysis We compute pointer information using (flow- and context-insensitive) Andersen-style pointer analysis. The issues regarding field sensitivity and heap cloning are not relevant in this example; they will be discussed in Section 2.3. The following points-to sets are found:

$$\begin{aligned} \mathsf{ptr}(o) &= o' \\ \mathsf{ptr}(\mathtt{buf}) &= \mathsf{ptr}(\mathtt{mbuf}) = \mathsf{ptr}(\mathtt{fbuf}) = o \end{aligned} \tag{1}$$

Phase 2: Full-Sparse SSA For this example, one region R is introduced to represent the singleton $\{o\}$, where o represents an abstract heap object created at line 13.

According to (1), R is aliased with `*buf`, `*mbuf` and `*fbuf`. Then loads, stores and callsites are annotated with μ and χ to make all 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`. Note that we have added $R_0 = \ldots$ in `freeBuf` as an implicit def as it receives its values from outside.

Any SSA algorithm can be used to derive the SSA form for each function individually, given in Figure 2(b).

Phase 3: SVFG This can be built on the SSA form. As shown in Figure 2(c), 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, 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 Figure 2(b)). 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.

2.3 Pre-Analysis

Initially, we perform a pre-analysis to compute the pointer/aliasing information in a program reasonably quickly and accurately. We use Andersen's inclusion-based pointer 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. All 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 at an allocation site are represented by one single abstract object.

After the pre-analysis is done, the points-to set $\mathtt{ptr}(v)$ for each pointer v is available. Each pointed-to target is either an abstract stack location or an abstract heap object. The points-to sets in Figure 2(a) are given in (1).

2.4 Building Full-Sparse SSA Form

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. The resulting def-use information remains sound for both single- and multi-threaded programs, because (1) pre-analysis is flow-insensitive and thus sound for single- and multi-threaded code, and (2) this sparse flow-sensitive refinement is intraprocedural and thus eliminates a spurious def-use only if both endpoints are in the same function.

Algorithm 1 Generating the regions for a program.

```
Procedure GenRegions
1  for each function f do
2      $Loc_f \longleftarrow \{\{v\} \mid v$ is a local variable declared in $f\}$;
3      $V_f \longleftarrow$ set of pointer dereferences of the form $*v$ in $f$;
4      $Non_f(*v) \longleftarrow$ subset of $\mathtt{ptr}(v)$, where $*v \in V_f$, representing
       all the nonlocal locations in $f$;
5      $R_f \longleftarrow Loc_f \cup (\bigcup_{*v \in V_f} \{Non_f(*v)\})$;
6      $REF_c$ ($MOD_c$) $\longleftarrow$ set of regions read (modified) at some
       loads (stores) at any callee function invoked directly/indi-
       rectly at a callsite $c$ in $f$ and visible in $f$;
7      $C_f \longleftarrow$ set of all callsites in $f$;
8      $Reg_f \longleftarrow R_f \cup (\bigcup_{c \in C_f} (REF_c \cup MOD_c))$;
```

Algorithm 2 Building full-sparse SSA form for a program.

```
Procedure BuildFullSparseSSA
9  GenMuChi;
10 BuildSSA;
Procedure GenMuChi
11 for each function f do
12     for each load $x = *y$ (store $*y = x$) $M$ in $f$ do
13         for each local region $L=\{v\}$, where $v \in \mathtt{ptr}(y) \setminus Non_f(*y)$ do
14             add $\mu(L)$ ($L = \chi(L)$) for $M$;
15         add $\mu(Non_f(*y))$ ($Non_f(*y) = \chi(Non_f(*y))$) for $M$;
16     for each callsite $c$ in $f$ (i.e., in $C_f$) do
17         for each region $R \in REF_c$ ($R \in MOD_c$) do
18             add $\mu(R)$ ($R = \chi(R)$) for $c$;
19 for each function f do
20     for each $\mu(R')$ ($R' = \chi(R')$) added for a statement $S'$ in $f$ do
21         for each $R = \chi(R)$ ($R = \chi(R)$ or $\mu(R)$) added for a
           different statement $S$ in $f$ such that $R' \cap R \neq \emptyset$ do
22             add $\mu(R)$ ($R = \chi(R)$) for $S'$;
```

The conversion to SSA requires nonlocal memory locations accessed by a pointer dereference expression $*v$ to be approximated. As a result, the uses at a load $\cdots = *v$ and the defs at a store $*v = \ldots$ are exposed by adding μ and χ functions. This must be done so that the resulting def-use chains are both accurate enough and amenable to fast traversal to satisfy the design objectives of SABER.

There are many solutions depending on how the memory is partitioned. At one extreme, FASTCHECK [2] 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 $\mathtt{ptr}(v)$ 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 f. Every local variable declared in f is in its own *local* region. It is assumed that all variables are distinctly named across the entire program. For every dereference expression $*v$ in f, all the nonlocal locations of f in $\mathtt{ptr}(v)$ are "collapsed" and denoted by one *nonlocal* region. These are dynamically created in f or its callees or declared in direct/indirect callers of f. Furthermore, SABER handles global variables like many static detectors such as FASTCHECK [2] and SPARROW [9]. All globals are represented by one single GLOBAL region. Heap objects that flow into GLOBAL directly or indirectly are not considered to leak, as further discussed in Section 2.6.

We apply GenRegions in Algorithm 1 to implement our memory model. Initially, for a given function f, R_f contains all the regions with its callsites ignored. Based on the pointer information discovered in pre-analysis, the interprocedural reference set REF_c and modification set MOD_c at a callsite c can be found in the standard manner, iteratively until a fixed-point is reached. Finally, Reg_f contains all regions accessed directly/indirectly in the function f.

Once all regions are identified, indirect defs and uses at loads/stores and callsites are added and the conversion to SSA can take place. As shown in BuildFullSparseSSA in Algorithm 2, GenMuChi introduces the μ and χ for loads, stores and callsites (lines 11 – 18), making sure that all aliased regions are included (lines 19 – 22). BuildSSA converts every function into SSA by using a standard SSA algorithm. For each $\mu(R)$ added at a callsite to a function, a store of the form $R = \dots$ is assumed to be available at the entry of the function. After the SSA conversion is done, R at the store has version 0 as it expects to "receive" values from its callers. This is illustrated using `freeBuf` in Figure 2(b).

In our memory model, aliases are recognised as overlapping regions and accounted for explicitly via μ's and χ's.

Consider our example in Figure 2. Based on (1), only one nonlocal region relevant to leak detection is found: $R = \{o\}$. Thus, the SSA form for the program is obtained as shown.

2.5 Building SVFG

Once a function is in SSA, the def-use chains in it 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 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 below. We write $\widehat{p_i}$ for the def site of an SSA variable p_i.

Table 2: Rules for building SVFG.

Rule	Statement (SSA)	Value-Flow Edges
ASN	$p_i = q_j$	$\widehat{p_i} \leftarrow \widehat{q_j}$
MU	$\mu(v_t)$ $p_i = *q_j$	$\widehat{p_i} \leftarrow \widehat{v_t}$
CHI	$*p_i = q_j$ $v_s = \chi(v_t)$	$\widehat{v_s} \leftarrow \widehat{q_j}$, $\widehat{v_s} \leftarrow \widehat{v_t}$
PHI	$p_i{=}\phi(q_j{,}q_k)$	$\widehat{p_i} \leftarrow \widehat{q_j}$, $\widehat{p_i} \leftarrow \widehat{q_k}$
CALL (at a callsite c for a callee g)	$\mu(v_m)$ $r_i{=}\mathrm{g}(\dots, a_k, \dots)$ $v_s = \chi(v_t)$	(1)$U_c^g(v_m) \leftarrow \widehat{\mu(v_m)}$, (2) $\widehat{\mu(v_m)} \xleftarrow{(_c^g} \widehat{v_m}$ (3)$FP(a_k) \xleftarrow{(_c^g} \widehat{a_k}$, (4) $\widehat{r_i} \xleftarrow{)_c^g} RET(r_i)$ (5) $\widehat{v_s} \xleftarrow{)_c^g} D_c^g(v_s)$, (6) $\widehat{v_s} \leftarrow \widehat{v_t}$

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 rules used are given in Table 2. By applying ASN, MU, CHI and PHI to a function, its intraprocedural def-use chains are added. In ASN, instead of linking $\widehat{q_j}$ to the use q_j at an assignment 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 CHI, $\widehat{v_s} \leftarrow \widehat{v_t}$ signifies a weak update to v_s using the old information in v_t and can be ignored if a strong update is possible (as described in Section 2.1). If v_0 is read in a function but passed from a callsite, then a $\mu(v_0)$ has been added to the callsite (Section 2.4).

Rule CALL, which looks complex, is also conceptually simple. There are six sub-rules. The middle two capture the value flows for the standard parameter passing and return for top-level pointers. The last one is there just like the case for CHI if a weak update on v_s is performed. The second last accounts for the "implicit" value returns for address-not-taken variables. Similarly, the first two model the "implicit" parameter passing for address-taken variables by treating the site of $\mu(v_m)$, denoted by $\widehat{\mu(v_m)}$, 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.

Note that `malloc` and `free` are special functions. An allocation site is marked as a *source* and a free site as a *sink*.

Let us now explain the technical details behind the first two and the second last sub-rules of CALL. $FP(a_k)$ stands for the corresponding formal parameter of a_k in SSA (version 0) and $RET(r_i)$ identifies the unique return SSA variable in g. Based on line 5 in Algorithm 1, REF_c^g and MOD_c^g denote the reference and modification sets made by this particular callee g, respectively, except that the variables contained are now in SSA. By construction, each SSA variable in REF_c^g has version 0 as it expects to "receive" values from its callers. Similarly, each SSA variable in MOD_c^g has the largest version for the underlying variable as it contains the final value defined in g. In Figure 2, `serialReadBuf` has two callsites at lines 3 and 9. We have $MOD_{\text{line } 3}^{\texttt{readbuf}} = \{R_1\}$ and $REF_{\text{line } 9}^{\texttt{freebuf}} = \{R_0\}$. The def "$R_0 = \dots$" added in `freeBuf` serves to receive its values from outside. The same is not done for `readBuf` since it is irrelevant in this example.

For a $v_s = \chi(v_t)$, we define $D_c^g(v_s) = \{R \mid R \in MOD_c^g, R \cap v_s \neq \emptyset\}$ to identify all regions in MOD_c^g that alias with region v_s. According to the second last sub-rule, an edge is added from every region in $D_c^g(v_s)$ to v_s (to realize the implicit value returns for address-taken variables). Similarly, for a $\mu(v_m)$, we define $U_c^g(v_m) = \{R \mid R \in REF_c^g, R \cap v_m \neq \emptyset\}$. We regard $\mu(v_m)$ as a pseudo formal parameter, $\widehat{\mu(v_m)}$, so that $\widehat{v_m}$ is first propagated to $\widehat{\mu(v_m)}$ and then to each region in $U_c^g(v_m)$, by the first two sub-rules. This realizes the implicit parameter passing for address-taken variables.

To achieve context-sensitive reachability analysis during leak detection, call and return edges are labelled with callsite information in the standard manner. In Rule CALL, the call edges are labelled with the open parenthesis $(_c^g$ and the return edges with the close parenthesis $)_c^g$. 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 [15, 16, 17].

Let us see how the SVFG in Figure 2(c) is built. The part corresponding to the source ① tracks the flow of o through the top-level pointers only into the sink ⑩, as is the case in Fastcheck [2]. The other part that tracks the flow of o' through the address-taken pointers, starting from the source ② and ending at the sink ⑨. Its edges are constructed as follows: ④ ← ② by the fifth sub-rule of CALL, ⑦ ← ⑤ ← ④ by the first two sub-rules of CALL, ⑧ ← ⑦ by Rule MU, and ⑨ ← ⑧ by the third sub-rule of CALL.

2.6 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 BDDs (Binary Decision Diagrams) 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 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 our unrealizable interprocedural flows of values as described in [15, 16, 17].

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 (free site) 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.

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 bugs 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,

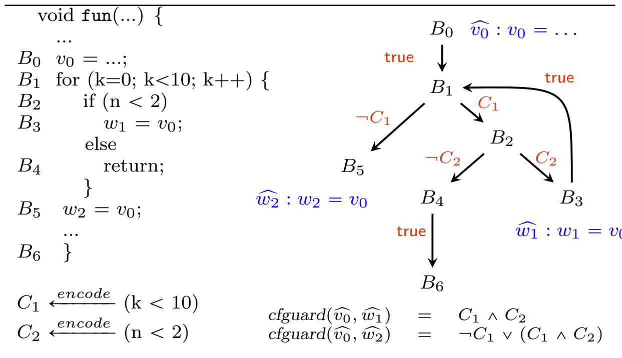
$$\mathsf{FREED}(src) = \bigvee_{tgt \in \mathcal{S}_{src}} \quad \bigvee_{P \in \mathit{vfpaths}(src,tgt)} \mathit{vfguard}(P) \qquad (2)$$

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 the use (site) $\widehat{q}$, denoted by *cfpaths*($\widehat{p}, \widehat{q}$). Thus,

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

There are two cases. If $(\widehat{p}, \widehat{q})$ is a call or return edge as marked in `CALL` given in Table 2, then $\mathit{cfguard}(\widehat{p}, \widehat{q}) = \mathsf{true}$ trivially since $|\mathit{cfpaths}(\widehat{p}, \widehat{q})| = 1$. 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,



```
      void fun(...) {
        ...
B0      v0 = ...;
B1      for (k=0; k<10; k++) {
B2         if (n < 2)
B3            w1 = v0;
           else
B4            return;
           }
B5       w2 = v0;
         ...
B6      }
```

$C_1 \xleftarrow{encode} (\mathrm{k} < 10)$

$C_2 \xleftarrow{encode} (\mathrm{n} < 2)$

$\mathit{cfguard}(\widehat{v_0}, \widehat{w_1}) = C_1 \wedge C_2$

$\mathit{cfguard}(\widehat{v_0}, \widehat{w_2}) = \neg C_1 \vee (C_1 \wedge C_2)$

B_0
$\widehat{v_0} : v_0 = \ldots$
true
B_1
true
$\neg C_1$
C_1
B_2
B_5
$\neg C_2$
C_2
$\widehat{w_2} : w_2 = v_0$
B_4
B_3
true
$\widehat{w_1} : w_1 = v_0$
B_6

Figure 3: Encoding paths with Boolean guards.

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

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:

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

There is one caveat as illustrated in Figure 3 using a portion of a backward slice regarding how the exit of a loop is handled. Note that $\mathit{cfguard}(\widehat{v_0}, \widehat{w_1}) = C_1 \wedge C_2$ 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 $\mathit{cfguard}(\widehat{v_0}, \widehat{w_1}) = \mathit{pguard}(Q) = C_1 \wedge C_2$. To compute $\mathit{cfguard}(\widehat{v_0}, \widehat{w_2})$, there are two 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, $\mathit{pguard}(Q_1) = \neg C_1$ and $\mathit{pguard}(Q_2) = C_1 \wedge C_2$. Finally, $\mathit{cfguard}(\widehat{v_0}, \widehat{w_2}) = \mathit{pguard}(Q_1) \vee \mathit{pguard}(Q_2) = \neg C_1 \vee (C_1 \wedge C_2)$.

Consider Figure 2(c). 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, $\mathsf{FREED}(o) = \mathsf{FREED}(o') = \mathit{Cond}$. So both are considered to leak on paths $\neg\mathit{Cond}$ since $\mathit{Cond} \not\equiv \mathsf{true}$.

3. THE SABER IMPLEMENTATION

We have implemented SABER in Open64, an open-source industry-strength compiler, at its IPA (interprocedural analysis) phase, as shown in Figure 4. IPA performs global 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

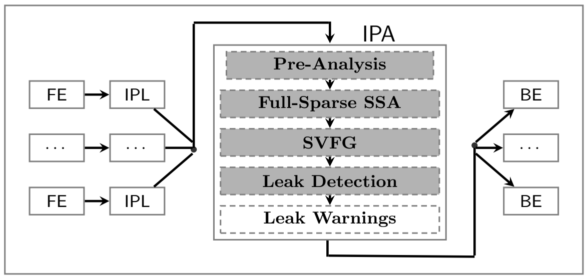


Figure 4: An implementation of Saber in Open64.

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.

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 [2], 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 [2].

To guarantee efficiency without losing much accuracy, the size of a backward slice is bounded by 100 nodes. A source is ignored if the limit is exceeded by its backward slice.

4. EXPERIMENTAL EVALUATION

We evaluate Saber using the 15 SPEC2000 C programs (620 KLOC) and five open-source applications (1.7 MLOC). We compare Saber with Contradiction [14] (data-flow analysis), Sparrow [9] (abstract interpretation), Clang, which stands here for its static analyzer (version checker-259) [8] (symbolic execution) and Fastcheck [2] (sparse value-flow analysis). Of these tools, only Fastcheck and Clang are publicly available. By using SPEC2000, a comparison between Saber and some other tools is made possible based on the data available in their papers.

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).

4.1 Practicality

Table 1 compares Saber with several other static leak detectors using the 15 SPEC2000 C programs. The data for Clang and Saber are produced in this work and the data for the others are obtained from their cited papers. Thus,

Table 3: Saber's bug counts and analysis times.

Program	Size (KLOC)	Time (secs)	Bug 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	20.88	40	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	100.0	22.03	8	2
httpd	128.1	10.65	0	0
icecast	22.3	5.54	12	5
sendmail	115.2	32.97	2	0
wine	1338.1	390.7	106	21
Total	2324.1	522.58	211	48

speed numbers should be regarded as rough estimates.

Saber reports 83 bugs among 103 leak warnings while Sparrow reports 81 among 96 warnings (without being able to compile `perlbmk` [9]). Fastcheck detects 59 bugs among 67 warnings. Both Contradiction and Clang find much fewer leaks. Saber detects consistently more bugs than the others while keeping its false positive rate at about 19% for SPEC2000. In addition, Saber achieves this level of accuracy by maintaining the same magnitude of speed with Fastcheck (at only a 3.7X slowdown) but is one order of magnitude faster than Sparrow (14.2X), Clang (25.6X) and Contradiction (34.1X). Overall, Saber is as accurate as Sparrow but is only slightly slower than Fastcheck.

To compare Saber further with Fastcheck and Clang, which are open-source tools, we have manually checked all their leak warnings and ours. In the case of Fastcheck, one of its authors graciously provided us their bug report. Saber succeeds in finding a superset of the bugs reported by each. Saber always detects no fewer bugs 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 used to enable leak detection. By analysing a function individually without considering its callers and callees, the information from outside (via its parameters and returns at callsites) is conservatively assumed to be unknown or symbolic. Thus, any objects created inside callees cannot be analyzed, thereby causing Clang to miss many bugs.

In addition to SPEC2000, we have also evaluated Saber using five open-source C applications. `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), `httpd-2.0.64` (an Apache HTTP server) and `sendmail-8.14.2` (a general-purpose internet email server). These five applications consist of 1.7 MLOC in total.

Table 3 summarises the accuracy and analysis times for the 15 SPEC2000 C programs and five applications. In `wine`, the largest in our suite, Saber finds 106 bugs with 21 false positives in about 390 secs, i.e., which is roughly the amount of time taken in compiling `wine` under "-O2". Overall, Saber

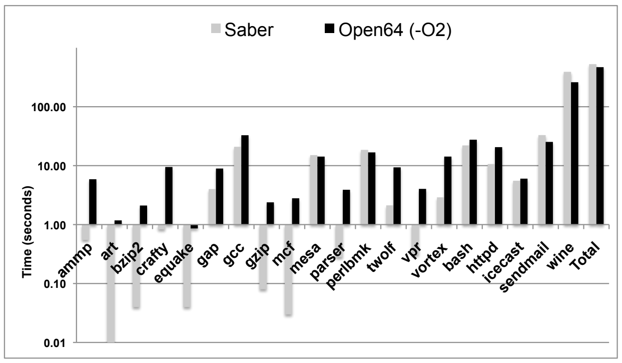


Figure 5: Comparing Saber's analysis times and Open64's compile times (under "-O2").

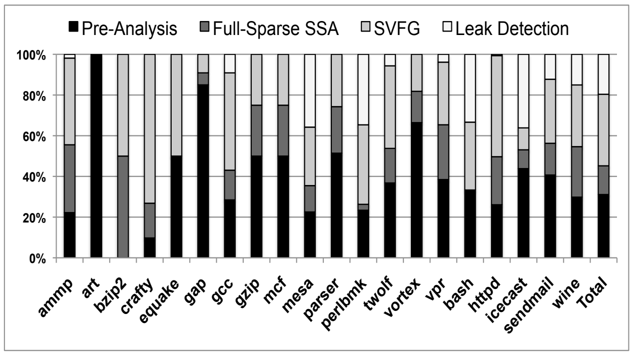


Figure 6: Percentage distribution of Saber's analysis times among its four phases.

finds a total of 211 leaks at a false positive rate of 18.5%. *To the best of our knowledge,* Saber *is the fastest memory leak detector scalable to millions of lines of code at this accuracy.*

4.2 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 Figure 5, 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 `gzip`, `vpr`, `art`, `mcf`, `equake`, `ammp` and `parser`, Saber's analysis times are significantly less than Open64's compile times. For some large programs like `perlbmk`, `wine` and `sendmail`, 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 $\mathcal{B}_{src}$ considered during full-path analysis are small.

Saber analyzes a program by going through its four phases in Figure 4. To understand their relative costs, Table 4 gives some statistics 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 variables in this work. The last five columns give the information concerning the leak detection phase, including the number of nodes in the SVFG and the sizes of its forward slices $\mathcal{F}_{src}$ built during some-path analysis and backward slices $\mathcal{B}_{src}$ built during all-path analysis (Section 2.6). 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 Figure 6, we can examine Saber's analysis times distributed among its four phases in our test suite. In total, their percentage distributions are pre-analysis (31.1%), full-sparse SSA (14.1%), SVFG (35.1%) and leak detection (19.7%). The pre-analysis and SVFG phases together dominate the analysis times for all the programs. The SSA phase seems to take some noticeable 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 3). 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 5.75% of the functions and 4.31% 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` 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 found to be either never freed or to reach GLOBAL during some-path analysis. In the case of `wine`, for example, Saber starts with 515 abstract objects. After the some-path analysis is done, there are 71 that are never freed and 268 that reach GLOBAL. So only 176 objects need to be further checked relatively more costly during all-path analysis.

4.3 Accuracy

We examine the causes for some interesting leaks reported by Saber to assess and understand further its accuracy. In the Fastcheck paper [2], the 15 SPEC2000 C programs and `bash` are also considered. When comparing Saber with Fastcheck, we refer to the bug report on these programs communicated to us by one of its authors. We also examine four representative scenarios with some leaks detected by Saber but missed by Fastcheck to highlight the importance of tracking value flows into address-taken variables.

As shown in Table 3, Saber finds 211 bugs with 48 false positives, achieving a false positive rate of 18.5% for our test suite. Let us consider SPEC2000 first. All the bugs (20) in `ammp` are conditional leaks, caused when functions do not free memory when returning on errors. All these bugs can also be found by Fastcheck as they require only value-flow

Table 4: Benchmark statistics ($\mathcal{F}'_{src}$ and $\mathcal{B}_{src}$'s stand for all forward and backward slices in a program, resp.).

Program	Characteristics					Leak Detection				
	#Functions	#Pointers	#Loads /Stores	#Allocation Sites	#Free Sites	#Nodes in SVFG	#Functions Included (%)		#SVFG's Nodes Included (%)	
							$\mathcal{F}_{src}$'s	$\mathcal{B}_{src}$'s	$\mathcal{F}_{src}$'s	$\mathcal{B}_{src}$'s
ammp	182	9829	1636	37	30	72362	11.54%	6.04%	2.38%	0.48%
art	29	600	103	11	1	2061	17.24%	3.45%	1.92%	0.20%
bzip2	77	1672	434	10	4	4943	5.19%	0.08%	0.43%	0.01%
crafty	112	11883	3307	12	16	56750	10.71%	2.89%	3.79%	1.07%
equake	30	1203	408	29	0	3071	13.33%	0.00%	1.57%	0.00%
gap	857	61435	16841	2	1	277614	0.23%	0.12%	0.01%	0.00%
gcc	2256	134380	51543	161	19	838373	14.43%	3.35%	8.59%	4.63%
gzip	113	3004	586	3	3	6048	6.19%	4.42%	4.55%	1.25%
mcf	29	1317	526	4	3	8160	48.28%	0.06%	2.42%	0.83%
mesa	1109	44582	17302	82	76	1427669	38.36%	8.66%	22.67%	10.10%
parser	327	8228	2597	1	0	29016	0.31%	0.00%	0.01%	0.00%
perlbmk	1079	54816	16885	148	2	698646	63.12%	21.07%	40.12%	16.18%
twolf	194	20773	8657	185	1	193074	48.45%	14.19%	30.38%	9.01%
vortex	926	40260	11256	9	3	146047	14.47%	0.41%	3.16%	1.05%
vpr	275	7930	2160	130	68	24814	41.82%	16.09%	7.18%	1.67%
bash	2700	17830	6855	112	58	32129	9.11%	2.33%	19.87%	11.23%
httpd	3000	60027	18450	21	18	176528	3.50%	0.17%	0.31%	0.05%
icecast	603	15098	9779	235	235	41474	37.25%	12.67%	33.07%	13.23%
sendmail	2656	107242	22191	296	136	824181	27.42%	15.49%	35.67%	8.90%
wine	77829	1330840	137409	515	231	2928148	8.75%	3.44%	10.59%	6.36%
Average							20.99%	5.75%	11.43%	4.31%

analysis for top-level pointers. All bugs reported in `gcc` are due to mishandling of strings. Most of these (with three inside loops) are related to calls to `concat`. For `mesa`, all the seven bugs found by Saber but missed by Fastcheck require value-flow analysis for address-taken variables. Some conditional leaks at a switch statement and some never-freed leaks are discussed below. Among the eight bugs 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 five applications, of which only `bash` is also analyzed by Fastcheck [2]. In `bash`, Fastcheck finds two of the eight bugs found by Saber. For the other six bugs, 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 bugs reported, three are related to mishandling 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 106 bugs in `wine`, with 71 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.

The false positives happened for several different reasons: not recognising 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 with regions (in `perlbmk` and `wine`).

Below we examine four representative scenarios, two from SPEC2000 (in `mesa`) and two from open-source programs (in `icecast` and `wine`). There are eight leaks, which all happen interprocedurally: six require value-flow analysis for address-taken variables and the remaining two can be found by analysing top-level pointers only.

Consider the code region from `mesa` given in Figure 7(a). 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 listed at the switch statement. In this case, the function returns directly at line 478, without freeing the two heap objects allocated earlier.

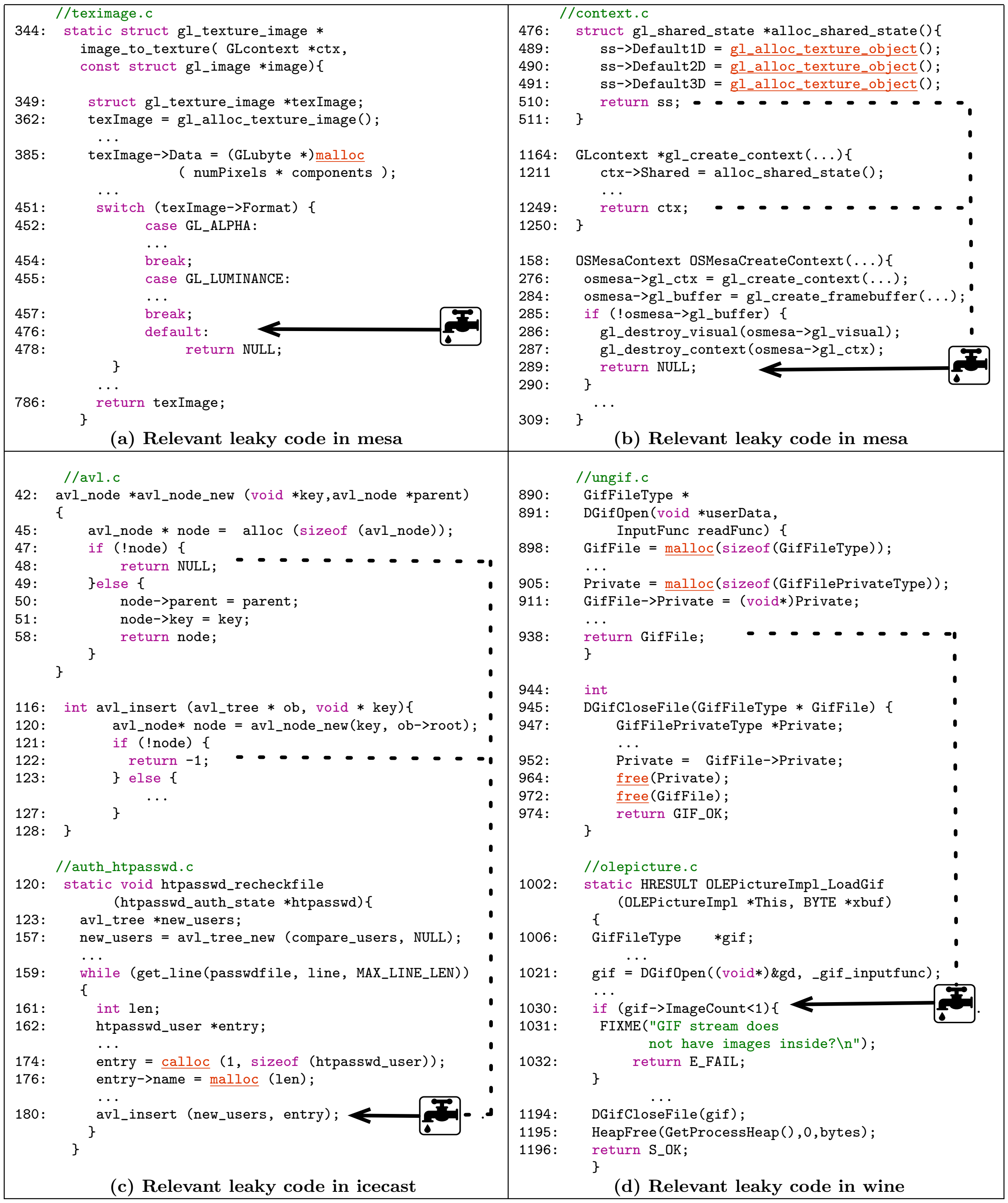
Consider the code region from `mesa` in Figure 7(b). At line 276, `gl_create_context` is invoked to create the three heap objects and assign them to three fields of `osmesa->gl_ctx` (lines 489 – 491). If the test `!osmesa->gl_buffer` at line 285 evaluates to true, these three objects leak since they are not freed in the call to `gl_destroy_context` at line 287.

Let us look at the two leaks in `icecast` as shown in Figure 7(c). 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 Figure 7(d), which are similar to the two illustrated earlier in our motivating example. In function `OLEPictureImpl_LoadGif`, `GifOpen` is called at line 1021 so that two heap objects are allocated at lines 898 and 905. One of the two objects 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, there is a test at line 1030 sitting between the two calls. The two objects are never freed when this test evaluates to true.

4.4 Limitations

Like nearly all static memory leak detectors, Saber is neither sound (by missing bugs) nor complete (by issuing false positives). Like Fastcheck, Saber bounds loops and recursion to at most one iteration and does not capture path



```
     //teximage.c
344:  static struct gl_texture_image *
        image_to_texture( GLcontext *ctx,
        const struct gl_image *image){

349:     struct gl_texture_image *texImage;
362:     texImage = gl_alloc_texture_image();
          ...
385:     texImage->Data = (GLubyte *)malloc
                    ( numPixels * components );
          ...
451:      switch (texImage->Format) {
452:            case GL_ALPHA:
                ...
454:            break;
455:            case GL_LUMINANCE:
                ...
457:            break;
476:            default:
478:                 return NULL;
             }
          ...
786:      return texImage;
        }
```

(a) Relevant leaky code in mesa

```
     //context.c
476:   struct gl_shared_state *alloc_shared_state(){
489:      ss->Default1D = gl_alloc_texture_object();
490:      ss->Default2D = gl_alloc_texture_object();
491:      ss->Default3D = gl_alloc_texture_object();
510:      return ss;
511:   }

1164:  GLcontext *gl_create_context(...){
1211      ctx->Shared = alloc_shared_state();
          ...
1249:     return ctx;
1250:  }

158:   OSMesaContext OSMesaCreateContext(...){
276:    osmesa->gl_ctx = gl_create_context(...);
284:    osmesa->gl_buffer = gl_create_framebuffer(...);
285:    if (!osmesa->gl_buffer) {
286:      gl_destroy_visual(osmesa->gl_visual);
287:      gl_destroy_context(osmesa->gl_ctx);
289:      return NULL;
290:    }
         ...
309:   }
```

(b) Relevant leaky code in mesa

```
      //avl.c
42:  avl_node *avl_node_new (void *key,avl_node *parent)
     {
45:      avl_node * node =  alloc (sizeof (avl_node));
47:      if (!node) {
48:          return NULL;
49:      }else {
50:          node->parent = parent;
51:          node->key = key;
58:          return node;
         }
     }

116:  int avl_insert (avl_tree * ob, void * key){
120:        avl_node* node = avl_node_new(key, ob->root);
121:        if (!node) {
122:          return -1;
123:        } else {
              ...
127:        }
128:  }

     //auth_htpasswd.c
120:  static void htpasswd_recheckfile
            (htpasswd_auth_state *htpasswd){
123:    avl_tree *new_users;
157:    new_users = avl_tree_new (compare_users, NULL);
        ...
159:    while (get_line(passwdfile, line, MAX_LINE_LEN))
        {
161:      int len;
162:      htpasswd_user *entry;
          ...
174:      entry = calloc (1, sizeof (htpasswd_user));
176:      entry->name = malloc (len);
          ...
180:      avl_insert (new_users, entry);
        }
      }
```

(c) Relevant leaky code in icecast

```
       //ungif.c
890:    GifFileType *
891:    DGifOpen(void *userData,
            InputFunc readFunc) {
898:    GifFile = malloc(sizeof(GifFileType));
        ...
905:    Private = malloc(sizeof(GifFilePrivateType));
911:    GifFile->Private = (void*)Private;
        ...
938:    return GifFile;
        }

944:    int
945:    DGifCloseFile(GifFileType * GifFile) {
947:        GifFilePrivateType *Private;
            ...
952:        Private =  GifFile->Private;
964:        free(Private);
972:        free(GifFile);
974:        return GIF_OK;
        }

        //olepicture.c
1002:   static HRESULT OLEPictureImpl_LoadGif
            (OLEPictureImpl *This, BYTE *xbuf)
         {
1006:    GifFileType     *gif;
            ...
1021:    gif = DGifOpen((void*)&gd, _gif_inputfunc);
         ...
1030:    if (gif->ImageCount<1){
1031:     FIXME("GIF stream does
                not have images inside?\n");
1032:        return E_FAIL;
         }
             ...
1194:    DGifCloseFile(gif);
1195:    HeapFree(GetProcessHeap(),0,bytes);
1196:    return S_OK;
         }
```

(d) Relevant leaky code in wine

Figure 7: Four scenarios with conditional leaks requiring value-flow analysis for address-taken variables.

correlations (except for tests against NULL). Thus, SABER shares the same limitations as FASTCHECK in these aspects. In addition, both tools, like many others, are not sound in handling pointer arithmetic by treating, for example, an occurrence of $x + e$ as an occurrence of x. SABER may also miss bugs due to imprecision in modelling the heap. SABER handles global variables similarly as in FASTCHECK and SPARROW. This is not sound since the leaks reachable by GLOBAL are not tracked.

Pre-analysis performed by using Andersen-style pointer analysis is fast but conservative. Its imprecision is improved with our intraprocedural flow-sensitive refinement during SSA construction and context-sensitive reachability analysis during leak detection. However, how to build more precise SVFGs more quickly is an ongoing effort.

5. RELATED WORK

There are a number of reported attempts on memory leak detection using static analysis [2, 7, 8, 9, 14, 20] or dynamic analysis [1, 4, 13, 22, 23]. 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 [2, 7, 9, 11, 14, 19]. SATURN [20] 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 when analysing some common programs [20]. BDDs are also used previously to represent and reason about program paths [18, 21]. CLANG [8] 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 [9] 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 [2] 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 analysing allocation sites whose values flow into top-level pointers only. CONTRADICTION [14] performs a backward data-flow analysis to disprove the presence of memory leaks. CLOUSEAU [7] 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, 4, 12] detect leaks by instrumenting and running a program based on test inputs. However, dynamic detectors tend to miss bugs although their false positive rates can be kept low. For example, we ran valgrind [12] on the same 15 SPEC2000 C programs using the reference inputs provided. More than 90% leaks reported by SABER cannot be detected.

Sparse Pointer Analysis Unlike iterative data-flow pointer analyses, their recent sparse incarnations [5, 6, 10, 18, 24] avoid propagating information unnecessarily guided by pre-computed def-use chains. Earlier, Hardekopf and Lin presented a semi-sparse flow-sensitive analysis [6]. 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 [5]. 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. [24] 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.

6. CONCLUSION

Memory leaks are common errors affecting many programs including OS kernels, desktop applications and web services. Some memory leaks can cause serious software reliability problems. 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 variables, SABER is effective at finding leaks in SPEC2000 and five open-source applications, by detecting a total of 211 leaks at a false positive rate of 18.5%.

7. ACKNOWLEDGMENTS

We thank Lonnie Princehouse of Cornell University for assisting us in comparing FASTCHECK with SABER, James Clause of the University of Dalaware for helping us with visualizing the bugs detected, and Jianxin Lai of the HP compiler team for his help with WHIRL SSA in Open64.

This work is supported by the Australian Research Council (ARC) grants, DP0987236 and DP110104628.

8. REFERENCES

[1] D. Bruening and Q. Zhao. Practical memory checking with Dr. Memory. In *CGO '11*.

[2] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI '07*.

[3] 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*.

[4] J. Clause and A. Orso. LEAKPOINT: pinpointing the causes of memory leaks. In *ICSE '10*.

[5] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO '11*.

[6] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL '09*.

[7] D.L. Heine and M.S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. *PLDI '03*.

[8] http://clang-analyzer.llvm.org/.

[9] Y. Jung and K. Yi. Practical memory leak detector based on parameterized procedural summaries. In *ISMM '08*.

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

[11] V.B. Livshits and M.S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. *FSE '03*.

[12] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *PLDI '07*.

[13] G. Novark, E.D. Berger, and B.G. Zorn. Efficiently and precisely locating memory leaks and bloat. *PLDI '09*.

[14] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. *SAS '06*.

[15] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*.

[16] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO '12*.

[17] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI '06*.

[18] Y. Sui, S. Ye, J. Xue, and P.C. Yew. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. *APLAS '11*.

[19] E. Torlak and S. Chandra. Effective interprocedural resource leak detection. In *ICSE '10*.

[20] Y. Xie and A. Aiken. Context-and path-sensitive memory leak detection. *FSE '05*.

[21] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *TOPLAS '07*.

[22] G. Xu, M.D. Bond, F. Qin, and A. Rountev. LeakChaser: helping programmers narrow down causes of memory leaks. In *PLDI '11*.

[23] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *ICSE '08*.

[24] 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*.