Region Analysis for Race Detection

Helmut Seidl and Vesal Vojdani

Lehrstuhl für Informatik II, Technische Universität München
Boltzmannstraße 3, D-85748 Garching b. München, Germany
{seidl,vojdanig}@in.tum.de

Abstract. Automatic race detection of C programs requires fast, yet sufficiently precise, analysis of dynamic memory. Therefore, we present a region-based pointer analysis which seeks to identify disjoint *regions* of dynamically allocated objects to ensure that write accesses to the same region are always protected by the same mutexes. Our approach has been implemented within the interprocedural analyzer of concurrent C programs GobLint and we have successfully applied it on code from the Linux kernel, such as the access vector cache. This code relies on a synchronized hash table where an array of doubly linked lists is protected by an array of locks.

1 Introduction

Writing multi-threaded code which both is correct and manipulates complicated data-structures can be cumbersome. Programmers of low-level software therefore mostly adhere to simple and conservative programming styles. Accordingly, dynamic shared data-structures are avoided whenever possible, and when dynamic allocation of memory is inevitable, one common idiom is to rely on *non-overlapping* data-structures and protect each of these *memory regions* by a dedicated lock. This occurs naturally when resources are maintained in hash-table-like data-structures, i.e., arrays of linked lists where each list is protected by its own lock as illustrated in Figure 1.

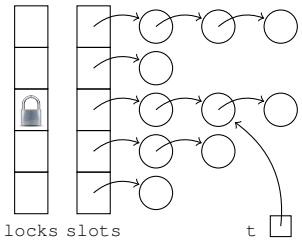

Fig. 1. Memory regions

There are different levels of granularity at which locking schemes for shared data-structures operate: at one extreme, an individual mutex is maintained for each data element separately, known as *per-element locking* [20]; at the other extreme, coarse-grained locking schemes use a single mutex to protect all data nodes allocated at a given point in the program. In between, there are subtler cases of medium-grained locking where certain dynamically allocated elements protect a bunch of other elements (not quite per-element), or elements allocated at a given point are not all protected by the same mutex (not quite coarse-grained). Here, we are concerned with the latter case. In many applications, we found that the dynamic data-structures protected by one mutex are disjoint from

```
typedef struct node { int data; struct node *next; } node;
node *even_list, *odd_list;

void insert(int data) {
    node *t = new(data);
    if (even(data)) { t→next = even_list; even_list = t; }
    else { t→next = odd_list; odd_list = t; } }

void even_worker() {
    node *t1 = even_list;
    while (t1 != NULL) {
        lock(even_mutex);
        access(t1→data);
        t1 = t1→next;
        unlock(even_mutex); } }
```

```
void odd_worker() {
    node *t2 = odd_list;
    while (t2 != NULL) {
        lock(odd_mutex);
        access(t2→data);
        t2 = t2→next;
        unlock(odd_mutex); } }
```

Fig. 2. Elements placed into linked lists

the data-structures protected by other mutexes. The *number* of protected disjoint data-structures, however, can be large. This is the case, e.g., for synchronized hash-tables where each bucket is protected by an individual mutex.

Consider the two-bucket hash-table in Figure 2 where elements allocated by the insert function end up in two distinct lists. The correctness of the locking scheme in this program hinges on the fact that the expressions `t1→data` and `t2→data` can never evaluate to the same address, i.e., they can never *alias*. We can be sure of this because the two lists are disjoint and thus closed under pointer reachability.

We call an analysis a *region analysis* if it infers a safe partitioning of the heap into disjoint regions. For region analysis, one could use sophisticated analyses to infer shapes of data-structures. Another approach would be to summarize dynamically allocated objects as *blobs* of memory associated with finitely many abstract locations such as allocation sites. While the first approach has difficulties scaling to larger programs, the second approach fails when elements allocated at the same program point end up in distinct data-structures protected by distinct mutexes, as in the above example.

We present a region analysis which is reasonably fast, yet sufficiently precise to deal with programs that manipulate disjoint heap regions. It identifies the set of static globals within the region accessed by local pointers. It also deals with arrays of regions by allowing regions to be indexed with symbolic index expressions. For the example above, the analysis would maintain that the two lists are disjoint, `t1` is pointing into the region of `even_list`, and `t2` is pointing into the region of `odd_list`.

Our region analysis can be extended to a race detection method by adding two components. First, a must alias analysis which provides information on which global address are definitely pointed to by a pointer variable, e.g., provided by [17]. Second, a symbolic lock set analysis which determines for every program

point a *representation*, which may involve symbolic address expressions, of the set of definitely held locks when reaching this program point.

2 Region Inference

For the purpose of this paper, we spell out our approach for a minimalistic programming model which is just rich enough to exhibit the key ideas of our analysis of multi-threaded programs using dynamic data-structures and arrays. At first, we restrict ourselves to programs which consist of only a single procedure represented by a finite control-flow graph where each edge is labeled with a basic operation s; in Section 3, we will extend to an interprocedural setting. We only track the values of local variables pointing into the global memory. The global memory is shared between different processes and consists of blocks, which either may be statically allocated at program start or dynamically allocated during program execution through some operator $\mathsf{new}\,\tau$ (for some type τ). For the moment, we rule out pointers into the stack as well as pointer arithmetic and assume that pointers always point to the beginning of blocks. In Section 5, we will add global arrays, and in Section 6, we indicate how the basic approach can be extended to work also in presence of (well-behaved) pointer-arithmetic as required for the analysis of, e.g., the Linux kernel API for doubly linked lists.

We assume that the frontend provides us with a normalized representation of assignments. For the beginning, we consider the following forms of expressions and assignments:

adr	$::= \mathbf{y}$	local pointer variable
	$\mid\ \&a$	static global address
$pexp$	$::= \mathbf{y} \rightarrow b$	dereferencing of pointers
val	$::= adr \mid \mathsf{null}$	pointer value
	$\mid\ \mathsf{new}(\tau)$	memory allocation
$pass$	$::= pexp = val;$	memory write
	$\mid\ \mathbf{y} = val; \mid\ \mathbf{y} = pexp;$	variable assignment

Let L and G denote the set of local pointer variables and the set of addresses of static global memory cells, respectively. Region analysis aims at inferring *potential reachability* between elements from $G \cup L$. Our analyzer therefore maintains for every program point an equivalence relation π on globals. Two elements $x_1, x_2 \in G$ are put into the same equivalence class when some memory cell is jointly reachable from both x_1 and x_2 through iterated field selection and dereferencing. Additionally, we maintain for every program point a function $\rho : L \rightarrow 2^{G \cup \{\bullet\}}$ mapping each local $\mathbf{y}$ to a set of globals identifying the region into which $\mathbf{y}$ may possibly point. The bullet $\bullet$ identifies the region of all thread-local dynamically allocated memory cells. When a thread allocates an object and initializes its fields, the object is seen as residing within this thread-local region until it is reachable from, or can itself reach, one of the global regions.

Equivalence relations have also been used for *may-alias* analysis [12]. There, two expressions are considered equivalent if they may denote the same address.

May-alias equivalence classes do not collapse when one is reachable from the other. On the other hand, while non-reachability *implies* non-equality, we cannot extract definite non-reachability information from non-equality. Thus, ensuring that pointers which traverse complicated structures may not alias is extremely difficult without the explicit notion of disjointness: one must precisely express the aliasing relationship, or all information about non-reachability is lost.

Here, an equivalence relation π is represented by the set of two-element subsets $\{x, y\}$ with $(x, y) \in \pi$ — implying that the *trivial* equivalence relation is represented by the empty set. Let $\mathbf{P}$ and $\mathbf{R}$ denote the set of all equivalence relations on G and the set of all maps from L to $2^{G \cup \{\bullet\}}$, respectively. Both sets form complete lattices for the orderings induced by the subset orderings on the set of two-element subsets of G and $G \cup \{\bullet\}$, respectively. In particular, for equivalence relations $\pi_1, \pi_2 \in \mathbf{P}$, the greatest lower bound $\pi_1 \sqcap \pi_2$ is given by the intersection of the sets of unordered pairs corresponding to π_1 and π_2, respectively; whereas the least upper bound $\pi_1 \sqcup \pi_2$ is the least equivalence relation containing all pairs from π_1 and π_2.

Using a suitable data-structure for partitions, the operations "$\sqcap$", "$\sqcup$" in $\mathbf{P}$ can be executed in polynomial time. Consider a pair $T = \langle \pi, \rho \rangle$ describing the current program state. We assume that all sets $\rho(\mathbf{y})$ are *closed* under π. We call a set X *closed* under the equivalence relation π, if $x \in X$ and $\{x, x'\} \in \pi$ implies that also $x' \in X$. For an arbitrary pair $\langle \pi, \rho \rangle$, let $\mathsf{cl}_\pi X$ denote the least set X' with $X \subseteq X'$ which is closed under π.

We now specify how a pair $T = \langle \pi, \rho \rangle$ describing the program state before an assignment s is transformed into a pair $\langle \pi', \rho' \rangle$ describing the program state after the assignment, i.e., we define the *abstract* meaning $[\![s]\!]^\sharp$ of the statement s. First, consider statements where local pointers are set:

$$\begin{aligned}
[\![\mathbf{y} = \&a]\!]^\sharp\, T &= \langle \pi, \rho \oplus \{\mathbf{y} \mapsto \mathsf{cl}_\pi\{\&a\}\} \rangle \\
[\![\mathbf{y} = \mathbf{y}']\!]^\sharp\, T &= [\![\mathbf{y} = \mathbf{y}' \to b]\!]^\sharp\, T = \langle \pi, \rho \oplus \{\mathbf{y} \mapsto \rho(\mathbf{y}')\} \rangle \\
[\![\mathbf{y} = \mathsf{null}]\!]^\sharp\, T &= \langle \pi, \rho \oplus \{\mathbf{y} \mapsto \emptyset\} \rangle \\
[\![\mathbf{y} = \mathsf{new}(\tau)]\!]^\sharp\, T &= \langle \pi, \rho \oplus \{\mathbf{y} \mapsto \{\bullet\}\} \rangle
\end{aligned}$$

where $\rho \oplus \{\mathbf{y}_i \mapsto X_i \mid i \in I\}$ is the function obtained from ρ by updating the image of $\mathbf{y}_i$ to X_i for all $i \in I$. Now, consider a write to memory through local pointers. In case either null or a pointer to a fresh memory block is written, the abstract state does not change:

$$[\![\mathbf{y} \to b = \mathsf{null}]\!]^\sharp\, T = [\![\mathbf{y} \to b = \mathsf{new}(\tau)]\!]^\sharp\, T = T$$

Finally, consider a write to memory of the form $\mathbf{y} \to b = \mathbf{y}'$. If neither $\rho(\mathbf{y})$ nor $\rho(\mathbf{y}')$ contain $\bullet$, then we join the equivalence classes of $\mathbf{y}$ and $\mathbf{y}'$:

$$\begin{aligned}
[\![\mathbf{y} \to b = \mathbf{y}']\!]^\sharp\, T = \langle \pi', \{\mathbf{y} \mapsto \mathsf{cl}_{\pi'}(\rho(\mathbf{y})) \mid \mathbf{y} \in L\} \rangle \qquad &\text{where} \\
\pi' = \pi \sqcup \{\{x, x'\} \mid x \neq x', x, x' \in \rho(\mathbf{y}) \cup \rho(\mathbf{y}')\} &
\end{aligned}$$

If the bullet is involved, but $\rho(\mathbf{y}), \rho(\mathbf{y}') \subseteq \{\bullet\}$, then simply $[\![\mathbf{y} \to b = \mathbf{y}']\!]^\sharp\, T = T$; however, when $\bullet \in \rho(\mathbf{y}) \cup \rho(\mathbf{y}') \not\subseteq \{\bullet\}$, we additionally consider all pointers that may point into the thread-local region denoted by $\bullet$. Let $Y = \{\mathbf{y}, \mathbf{y}'\} \cup \{\mathbf{y}'' \in L \mid \bullet \in \rho(\mathbf{y}'')\}$ and $X = \bigcup\{\rho(\mathbf{y}'') \mid \mathbf{y}'' \in Y\} \backslash \{\bullet\}$. We join all globals from X into one equivalence class to which all variables from Y may now point:

$$
\begin{aligned}
[\![\mathbf{y} \to b = \mathbf{y}']\!]^\sharp\, T = \langle \pi', \rho' \rangle \qquad & \text{where} \\
& \pi' = \pi \sqcup \{\{x, x'\} \mid x \neq x', x, x' \in X\} \\
& \rho' = \{\mathbf{y} \mapsto \mathsf{cl}_{\pi'}(\rho(\mathbf{y})) \mid \mathbf{y} \notin Y\} \cup \{\mathbf{y}'' \mapsto \mathsf{cl}_{\pi'} X \mid \mathbf{y}'' \in Y\}
\end{aligned}
$$

For proving the soundness of the analysis, we rely on a small-step operational semantics of heap-manipulating programs. Since we have currently ruled out procedures, the concrete program state when reaching a program point u consists of a pair $\sigma = \langle \mu, \lambda \rangle$ where λ maps the local pointers to the start addresses of blocks and μ describes the current global memory. We represent the memory μ by a map which assigns a value to every address-field pair (l, b). Type-safety requires that l is the address of a block in the global memory of struct type τ which has a field b. For convenience, we assume that every field of pointer type which has not yet been initialized, holds the value null.

In μ, the address l_1 is *reachable* from the address l_2 iff l_2 can be obtained from l_1 by repeated field selection and dereferencing. A *region* in μ is a set R of addresses in μ such that every $l_1 \in R$ satisfies the condition: $l_2 \in R$ whenever $\mu(l_1, b) = l_2$ for some field name b of the struct at address l_1. This definition implies that the set of regions of μ form a *partition* of the addresses in μ. In particular, no address in the region R is reachable from any address outside the region R.

Assume that the concrete program state $\sigma = \langle \mu, \lambda \rangle$ induces a partition $\Pi = \{R_1, \ldots, R_m\}$ of the addresses in μ. Then σ is in the *concretization* of the abstract state $T = \langle \pi, \rho \rangle$, i.e., $\sigma \in \gamma(T)$, iff the following holds:

1. Whenever $\{x, x'\} \notin \pi$ for global static addresses $x \neq x'$, then x and x' are not in the same region of μ.
2. Whenever $x \notin \rho(\mathbf{y})$, then x and $\lambda(\mathbf{y})$ are not in the same region of μ.
3. Whenever $\rho(\mathbf{y}) = \emptyset$, then $\lambda(\mathbf{y})$ equals null.

This implies that if $\rho(\mathbf{y}) = \{\bullet\}$, then all memory cells reachable from $\lambda(\mathbf{y})$ are definitely not reachable from globals and thus not accessible from other threads. Accordingly, write accesses through $\mathbf{y}$ need not be protected. If on the other hand, $\rho(\mathbf{y})$ contains a global static address, the address of $\mathbf{y}$ must be considered as *published*, i.e., possibly accessible for other threads. The set of static global addresses occurring in ρ (and π) can be considered as the set of *possible owners* of a region for which locks should be provided. The following theorem states that our definitions of the abstract transformers for basic program statements are sound.

Theorem 1 (Soundness of Transfer Functions). *Let s denote a program statement and T denote an abstract state. If $\sigma \in \gamma(T)$ and σ' denotes the concrete program state obtained from σ by the execution of s, then $\sigma' \in \gamma([\![s]\!]^\sharp T)$.* □

3 Interprocedural Analysis

In this section we present an interprocedural formulation of the region analysis. We model communication between procedures by assuming that every function has the same set L of local variables and that all locals of the caller are passed by value to the callee; however, in our simplified setting, we only pass locals into procedures but do not return them back. Thus, the effect of a procedure call is limited to possible collapses within the partition of globals and the possible joining of thread-local data structures with some global regions. In order to deal with the latter, we extend the points-into map ρ for local pointer variables with an extra variable $\diamond$ representing the thread-local data structures before the call. The abstract transformer $\mathsf{enter}^\sharp$ initializes the abstract state at procedure entry based on the abstract state before the call:

$$\mathsf{enter}^\sharp(\langle \pi, \rho \rangle) = \langle \pi, \rho \oplus \{\diamond \mapsto \{\bullet\}\} \rangle$$

While analyzing a procedure q, updates through pointers into thread-local memory may result in globals being added to the region tracked by $\diamond$ (just as for any other variable with $\bullet$ in its points-into set). At procedure exit, the local variables of the called procedure q are removed, while the points-into information accumulated by $\diamond$ are added to every local $\mathbf{y}$ of the caller which before the call may have pointed into the thread-local region. Assume that $T_1 = \langle \pi_1, \rho_1 \rangle, T_2 = \langle \pi_2, \rho_2 \rangle$ are the abstract states before the call and at procedure exit, respectively. Then this combination is achieved by the function $\mathsf{combine}^\sharp$:

$$\begin{aligned} \mathsf{combine}^\sharp(T_1, T_2) &= \langle \pi_2, \rho \rangle \qquad \text{where} \\ \rho &= \{z \mapsto \mathsf{cl}_{\pi_2}(\rho_1(z)) \mid z \in L \cup \{\diamond\}, \bullet \notin \rho_1(z)\} \cup \\ &\quad \{z \mapsto \mathsf{cl}_{\pi_2}(\rho_1(z) \cup \rho_2(\diamond)) \mid z \in L \cup \{\diamond\}, \bullet \in \rho_1(z)\} \end{aligned}$$

The abstract functions $\mathsf{enter}^\sharp$ and $\mathsf{combine}^\sharp$ allow us to apply general frameworks for interprocedural analysis [26]. Here, we follow the approach advocated, e.g., by Cousot [5], which relies on *partially tabulating* the abstract value tables of called procedures. A multi-threaded variant of this approach [25] has been implemented by the analyzer Goblint [28]. The analyzer solves a constraint system for the abstract values returned by the summary function for f when called on abstract values a. Given a complete lattice $\mathcal{L}$ of abstract values, abstract transformers $[\![s]\!]^\sharp$ for basic statements, and abstract transformers $\mathsf{enter}^\sharp$ and $\mathsf{combine}^\sharp$ for parameter passing and function return, the constraint system is set up as follows:

$$\begin{array}{ll} \langle v, a \rangle \sqsupseteq a & \text{for a function entry point } v \\ \langle v, a \rangle \sqsupseteq [\![s]\!]^\sharp\ (\langle u, a \rangle) & \text{for edge } (u, v) \text{ with statement } s \\ \langle v, a \rangle \sqsupseteq \mathsf{combine}^\sharp(\langle u, a \rangle, \langle \mathsf{ret}_f, \mathsf{enter}^\sharp(\langle u, a \rangle) \rangle) & \text{for edge } (u, v) \text{ calling } f() \end{array}$$

where $a \in \mathcal{L}$, f denotes functions with return point ret_f, and u, v are program points. For a program point v of a function g, the variable $\langle v, a \rangle$ of the constraint system represents the abstract value attained at v in a call to g where evaluation of the body of g starts with the abstract value a. The soundness of the least

solution of this constraint system instantiated to our region analysis follows from Theorem 1 and [5, 14]:

Theorem 2 (Soundness of Region Analysis). *Assume that $\varphi\langle v, a\rangle$, for program point v of a procedure f and abstract state a, is the least solution of the constraint system over the complete lattice $\mathcal{L}$. Let $\varphi\langle v, a_e\rangle = \langle \pi, \rho\rangle$, and assume that the pair $\sigma_e = \langle \mu_e, \lambda_e\rangle$ of a heap μ_e and assignment λ_e of locals is in the concretization of a_e, i.e., $\sigma_e \in \gamma(a_e)$. Moreover, assume that R_e is the set of thread-local memory cells at procedure entry, i.e., the set of addresses which can only be reached from the locals in σ_e.*

Then every same-level execution starting in σ_e at the entry point of f and reaching program point v in state $\sigma = \langle \mu, \lambda\rangle$ satisfies the following properties:

- $\sigma \in \gamma(\langle \pi, \rho\rangle)$;
- *For every global x, if x is reachable from an address in R_e (w.r.t. μ), or an address in R_e is reachable from x (w.r.t. μ), then $\&x \in \rho(\diamond)$.* □

The given constraint system may be huge depending on the complete lattice of the analysis. *Local fixpoint iteration* is a general technique to partially explore large (or possibly infinite) systems of constraints [7]. Starting from a subset Y of *interesting* unknowns, local fixpoint iteration explores only those other unknowns which may *contribute* to the values of unknowns from Y. This technique is well-suited if the interesting values can be computed by consulting only a small (though possibly unknown) fraction of the constraint variables. This is the case in our application. Here, fixpoint iteration starts with the set $Y = \{\langle \mathsf{ret}_{\mathsf{main}}, \mathsf{enter}^\sharp\, a\rangle\}$ if main is the start function of the thread currently under consideration, and the abstract value a describes the program state before program execution [7]. Local fixpoint iteration then will trigger the evaluation of all pairs $\langle v, \mathsf{enter}^\sharp a'\rangle$ where v is the program point of a procedure which (during fixpoint iteration) is called for the abstract program state a'. In our experiments with the analyzer Goblint, we found that the number of different calls of the same procedure is mostly quite small.

4 Relating Locks and Regions

In order to relate accessed regions of the global memory with acquired locks, we can rely on any analysis providing *must-alias* information for static global addresses. For clarity of presentation, we just consider the simplest instance of such an analysis, which tracks conjunctions of equalities of the form $\mathbf{y} \doteq x$ where $\mathbf{y} \in L$ is a local pointer variable and $x \in L \cup G$ is either a local pointer variable or a global static address. Such a domain has been suggested in [17] where efficient algorithms for the basic operations have been presented.

Let $\mathbf{E}$ denote the lattice of equalities. Technically, each element $\phi \in \mathbf{E}$ either is equivalent to false or is equivalent to a satisfiable finite conjunction of equalities. We write $\phi \models (x \doteq x')$ if the equality $x \doteq x'$ is logically implied by ϕ. The ordering on $\mathbf{E}$ is given by logical implication, i.e., $\phi \sqsubseteq \phi'$ iff either $\phi = \mathsf{false}$ or

both ϕ and ϕ' are different from false, and $\phi \models (x \doteq x')$ for every equality $x \doteq x'$ in ϕ'. Thus, the greatest lower bound of ϕ_1, ϕ_2 is given by their conjunction $\phi_1 \wedge \phi_2$, whereas the least upper bound of two satisfiable conjunctions ϕ_1, ϕ_2 is equivalent to the conjunction of all equalities $x \doteq x'$ which are both implied by ϕ_1 and ϕ_2. Here, we consider the abstract functions for procedure calls. According to our assumption, all locals are passed as actual parameters to called procedures. The locals of the caller, on the other hand, are not affected by the changes to locals of the callee. This means that the abstract functions $\mathsf{enter}^\sharp_E, \mathsf{combine}^\sharp_E$ for procedure calls are defined by:

$$\mathsf{enter}^\sharp_E\, \phi = \phi \qquad \mathsf{combine}^\sharp_E(\phi_1, \_) = \phi_1$$

As a third component, our analysis requires information about the set of locks which are definitely held when reaching a program point. For the moment, every lock is identified by static addresses or addresses pointed at by local pointers. For every reachable program point u (in every analyzed invocation of a procedure), our analysis therefore identifies a finite subset S of descriptions of locks which are definitely held when reaching u (in the given invocation). Let $\mathbf{S}$ denote the set of finite subsets of global static addresses of locks. Since we are interested in *definite* information, finite sets of lock address expressions are ordered by the *superset* relation.

While region and must-alias analysis are independent, the analysis of sets of definitely held locks may profit from the results of both. The must-alias analysis is applied to identify all address expressions which denote the acquired lock, the may-alias information which we infer from the region information, helps to narrow down the set of locks which may no longer be held after releasing a lock. More precisely, assume that $T = \langle \pi, \rho \rangle$ is an abstract description of memory regions. We infer non-equality information as follows. If $\{x, x'\} \notin \pi$ for two pointer expressions x, x', then $x \neq x'$ for every program state $\langle \mu, \lambda \rangle$ in the concretization of π. Likewise, if $x \notin \rho(\mathbf{y})$, then also $\lambda(\mathbf{y}) \neq x$. Finally, if $\rho(\mathbf{y}) \cap \rho(\mathbf{y}') = \emptyset$ while $\rho(\mathbf{y}) \cup \rho(\mathbf{y}') \neq \emptyset$, then also $\lambda(\mathbf{y}) \neq \lambda(\mathbf{y}')$. We denote these facts by $T \models (x \neq x')$, $T \models (\mathbf{y} \neq x)$ and $T \models (\mathbf{y} \neq \mathbf{y}')$, respectively.

Assume that the current program state $T = \langle \pi, \rho, \phi, S \rangle$ consists of the partition of globals π, the points-into information ρ, the conjunction of must-equalities ϕ, and the lock set S. Then the sets of definitely held locks after operations lock and unlock for locks inside static structs are defined by:

$$\begin{aligned} [\![\mathsf{lock}(\&(z \to b))]\!]^\sharp_S T &= S \cup \{\&(x \to b) \mid x \in G, \phi \models z \doteq x\} \\ [\![\mathsf{unlock}(\&(z \to b))]\!]^\sharp_S T &= S \setminus \{\&(x \to b) \mid \neg(\pi \models z \neq x)\} \end{aligned}$$

for $z \in L \cup G$, respectively. When entering or leaving a procedure, the set of definitely held locks does not change. Therefore, we have:

$$\mathsf{enter}^\sharp_L\, S = S \qquad \mathsf{combine}^\sharp_L(\_, S_2) = S_2$$

```
struct list { int key; int data; struct list *next; };
struct list *slots[512];
spinlock_t   locks[512];

struct list *insert(int key, int data) {
    struct list *t; int hv = hash(key);
    spin_lock(&locks[hv]);
    t = slots[hv];
    if (t == NULL) {
        slots[hv] = new_list(key, data); goto fd; }
    while(1) {
        if (t→key == key) {
            t→data = data; goto fd; }
        if (t→next == NULL) {
            t→next = new_list(key, data); goto fd; }
        t = t→next; }
fd: spin_unlock(&locks[hv]);
    return t; }
```

Fig. 3. Simplified `insert`-function.

5 Extension with Arrays

So far, our analysis is able to deal with dynamic data structures and a fixed finite set of mutexes. In the next step, we extend this base approach to global data structures which may contain arrays and thus also arrays of mutexes.

Example 1. Figure 3 shows a simplified version of the insert-function from the access vector cache of Security Enhanced Linux.[1] At every program point, at most one lock is held which is taken from a possibly large set of locks contained in the array `locks`. For a sound data-race analysis of the function `insert`, it does not suffice to verify that *some* lock from this array is held when the hash map is modified. Instead, it also must check the (statically unknown) *index* of the lock coincides with the index of the list in `slots`. □

We now extend our core language by additionally allowing arrays within global shared data structures. Here, we consider non-nested arrays only. The address of a memory cell from a *static* global data structure with arrays is identified by $\&a[i]$ where i is an index. Accordingly, we consider address expressions of the form $\&a[e]$ where e is a side-effect free index expression depending on **int**-variables only. Furthermore, we extend our notion of abstract heap partitions π and points-into maps ρ. Besides sets of two-element sets, we now also allow

[1] The most notable simplification is the use of singly linked lists instead of the doubly linked lists from the Linux kernel; however, since our technique is based on a conservative partitioning of the heap into disjoint regions, dealing with doubly linked lists and even structured use of pointer arithmetic posed no significant further challenge.

singleton sets $\{\&a\}$ in partitions. Such a singleton indicates that different entries of the array $\&a$ may belong to the same memory region. We thus consider the set $\mathbf{P}$ of abstract heap partitions π with the following properties:

1. If $\{x, y\}, \{y, z\} \in \pi$ for $x \neq z$, then $\{x, z\} \in \pi$.
2. If $\{\&a, x\} \in \pi$, then also $\{\&a\} \in \pi$.
3. If $\{\&a\} \in \pi$, then $\&a[e]$ does not occur in π.
4. For the same array $\&a$, π may have at most one address expression e with $\&a[e]$ occurring in π.

We could have allowed multiple index expressions e_i referring to the same array $\&a$ as long as all e_i definitely evaluate to distinct values. In our experiments, the restriction to a single expression, however, has always been sufficient. The partial ordering on $\mathbf{P}$ is given by $\pi_1 \sqsubseteq \pi_2$ iff the following holds:

1. If $\{\&a[e], x\} \in \pi_1$ then $\{\&a[e], x\} \in \pi_2$ or $\{\&a\}, \{\&a, x\} \in \pi_2$.
2. If $\{x, y\} \in \pi_1$ where neither x nor y contains an index expression, then also $\{x, y\} \in \pi_2$.

Thus, e.g., for $\pi_1 = \emptyset$, $\pi_2 = \{\{\mathbf{p}, \&a[\mathbf{i}]\}\}$, $\pi_3 = \{\{\&a\}, \{\mathbf{p}, \&a\}\}$, $\pi_1 \sqsubseteq \pi_2 \sqsubseteq \pi_3$.

Accordingly, we now consider points-into maps ρ where a set X occurring as the image of a local (or $\diamond$) satisfies the following additional restrictions:

1. If $\&a[e], \&a[e'] \in X$, then $e \equiv e'$;
2. If $\&a \in X$ then for every e, $\&a[e] \notin X$

where the ordering on two such sets is the natural extension of $\emptyset \sqsubseteq \{x\}$ for all x, and $\{\&a[e]\} \sqsubseteq \{\&a\}$.

Also, we extend the closure operation cl_π such that $\mathsf{cl}_\pi X$ for a set X of global static address expressions or $\bullet$, now additionally replaces an indexed expression $\&a[e]$ with $\&a$ whenever $\{\&a\} \in \pi$. Likewise, we extend the domain of must equalities and finite lock sets to address expressions containing indexing. The occurring index expressions may depend on **int**-variables; however, we here ignore definite equalities between **int**-variables. Thus, we consider two index expressions e_1, e_2 as *definitely* equal only if they are *syntactically* equal. Technically, this allows us to use a similar domain for must equalities and lock sets as in section 4 — only that we now additionally consider indexed static addresses $\&a[e]$ instead of static addresses $\&a$ alone.

This simplistic setting is still able to deal with increments or decrements of **int**-variables. Accordingly, our analysis will track assignments to **int**-variables $\mathbf{i}$ of the form $\mathbf{i} = \mathbf{i} + c$ for $c \in \mathbf{Z}$ whereas all other assignments to $\mathbf{i}$ are approximated by the *non-deterministic* assignment $\mathbf{i} = ?$ which is meant to assign to $\mathbf{i}$ an *unknown* value. The effect of the assignment $\mathbf{i} = \mathbf{i} + c$ on a triple $T = \langle \pi, \phi, S \rangle$ consists in substituting $\mathbf{i}$ in all index expressions occurring in T with $\mathbf{i} - c$. The effect of the assignment $\mathbf{i} = ?$ on the other hand, assigns an unknown value to $\mathbf{i}$ and thus must remove all occurrences of $\mathbf{x}_i$ from T. For a partition π, $\mathsf{delete}(\pi, \mathbf{i})$ replaces all expressions $\&a[e]$ where $\mathbf{i}$ occurs in e with $\&a$ (if there are any) and adds the set $\{\&a\}$ (given that there are any). For a points-into map ρ, $\mathsf{delete}(\pi, \mathbf{i})$

replaces in every image $\rho(z)$ elements $\&a[e]$ where $\mathbf{i}$ occurs in e with $\&a$. For component ϕ, $\mathsf{delete}(\phi, \mathbf{i})$ removes all equalities involving $\mathbf{i}$. Likewise for S, $\mathsf{delete}(S, \mathbf{i})$ removes all lock expressions $\&a[e].b$ where $\mathbf{i}$ occurs in e.

$$\begin{aligned}[\![\mathbf{i} = \mathbf{i} + c]\!]^\sharp T &= T[\mathbf{i} - c/\mathbf{i}] \\ [\![\mathbf{i} = ?]\!]^\sharp T &= \langle \mathsf{delete}(\pi, \mathbf{i}), \mathsf{delete}(\rho, \mathbf{i}), \mathsf{delete}(\phi, \mathbf{i}), \mathsf{delete}(S, \mathbf{i}) \rangle\end{aligned}$$

The effects of assignments involving local pointers and global memory, are defined componentwise on the first three components, while the set of definitely held locks remains unchanged. We omit the details but instead apply the technique to a typical example.

Example 2. Assume we start the execution of the insert-function from Figure 3 with the abstract value $T_0 = \langle \emptyset, \{\diamond \mapsto \{\bullet\}, \mathbf{t} \mapsto \emptyset\}, \mathsf{true}, \emptyset \rangle$. After having called `spin_lock()` and reaching the *while*-loop, we have:

$$\begin{aligned}T_1 &= \langle \emptyset, \rho_1, \phi_1, S_1 \rangle \qquad \text{where} \\ \rho_1 &= \{\diamond \mapsto \{\bullet\}, \mathbf{t} \mapsto \{\&\mathtt{slots}[\mathbf{hv}]\}\} \\ \phi_1 &= \mathbf{t} \doteq \&\mathtt{slots}[\mathbf{hv}] \\ S_1 &= \{\&\mathtt{locks}[\mathbf{hv}]\}\end{aligned}$$

although the precise value of $\mathbf{hv}$ is unknown. Inside the loop the must-equality $\mathbf{t} \doteq \&\mathtt{slots}[\mathbf{hv}]$ is lost, while the region information as well as the lock set are preserved. Unlocking resets the set of held locks to $\emptyset$. □

Our analysis can be enhanced by jointly performing constant propagation or, more generally, any analysis of **int** variables which provides us with more precise information about how index expressions are related. Such information could be provided, e.g., by Karr's analysis of affine equalities [13, 16].

While the complete lattice for the combined analysis of regions, must equalities and abstract lock sets in presence of arrays is no longer finite, it still satisfies the ascending chain condition. In order to apply the interprocedural framework from Section 3, we generalize the functions $\mathsf{enter}^\sharp$ and $\mathsf{combine}^\sharp$ for abstract parameter passing and procedure return from the last sections. Additionally, we now must track the values of local **int** variables. We could do so by additionally maintaining, e.g., affine must equalities between these. Here, we prefer a simpler analysis which just tracks the set of local **int** variables which may have changed their values since procedure entry. Assume that before the call, we have the abstract state $T = \langle \pi, \rho, I, \phi, S \rangle$ where π, ρ, ϕ, and S are as before and I now denotes a set of **int** variables whose value possibly has changed since procedure entry. When entering a newly called procedure, we initialize this set to $\emptyset$. We define:

$$\begin{aligned}\mathsf{enter}^\sharp \langle \pi, \rho, I, \phi, S \rangle &= \langle \pi, \rho_1, \emptyset, \phi, S \rangle \quad \text{where} \\ \rho_1 &= \rho \oplus \{\diamond \mapsto \{\bullet\}\}\end{aligned}$$

Likewise, at procedure exit, the local variables of the called procedure q must be removed. Also all equivalences $\{x, \&a[e]\}$ in the returned must be collapsed to

$\{x, \&a\}$ for index expressions e depending on **int**-variables which have changed their values. This is achieved by:

$$\begin{aligned}
\mathsf{combine}^\sharp(\langle \pi_1, \rho_1, I_1, \phi_1, S_1\rangle, \langle \pi_2, \rho_2, I_2, _, S_2\rangle) \quad &= \quad \langle \pi, \rho, I_1, \phi_1, S\rangle \quad \text{where} \\
\pi &= \mathsf{delete}(\pi_2, I_2) \\
\rho &= \{z \mapsto \mathsf{cl}_\pi(\rho_1(z)) \mid z \in L \cup \{\diamond\}, \bullet \notin \rho_1(z)\} \cup \\
&\quad \{z \mapsto \mathsf{cl}_\pi(\rho_1(z) \cup \rho_2(\diamond)) \mid z \in L \cup \{\diamond\}, \bullet \in \rho_1(z)\} \\
S &= \mathsf{delete}(S_2, I_2)
\end{aligned}$$

Here, the calls to $\mathsf{delete}()$ for a set I of **int** variables abbreviate repeated application of $\mathsf{delete}()$ for each element $\mathbf{i} \in I$.

Example 3. Consider the insert-function from Figure 3. Assume that at the program point before the call to this function we have the abstract state: $T_0 = \langle \emptyset, \{\diamond, \mathbf{t} \mapsto \{\bullet\}\}, \emptyset, \mathsf{true}, \emptyset\rangle$. Then $\mathsf{enter}^\sharp(T_0) = T_1$ is the abstract value for the start point of the corresponding abstract call to the function `insert()` where:

$$\begin{aligned}
T_1 &= \langle \emptyset, \rho_1, \emptyset, \phi_1, \emptyset\rangle \qquad \text{where} \\
\rho_1 &= \{\diamond \mapsto \{\bullet\}, \mathbf{t} \mapsto \{\&\texttt{slots}[\mathbf{hv}]\}\} \\
\phi_1 &= \mathbf{t} \doteq \&\texttt{slots}[\mathbf{hv}]
\end{aligned}$$

At the program point before the lock operation, we have $T_2 = \langle \emptyset, \rho_1, \{\mathbf{hv}\}, \phi_1, \emptyset\rangle$. After locking, we thus have $T_3 = \langle \emptyset, \rho_1, \{\mathbf{hv}\}, \phi_1, \{\&\texttt{locks}[\mathbf{hv}]\}\rangle$ — implying that the elements accessed through the pointer $\mathbf{t}$ belong to the region $\texttt{slots}[\mathbf{hv}]$ and that these accesses are protected by the corresponding lock $\texttt{locks}[\mathbf{hv}]$. At function exit, we finally arrive at $T_4 = \langle \emptyset, \rho_1, \{\mathbf{hv}\}, \phi_1, \emptyset\rangle$. Combining this state with the state T_0 before the call will recover the set of possibly modified **int** variables as well as the must equalities before the call. In the example, we just recover the abstract state T_0. □

6 Analyzing the Linux Kernel

We have implemented our analysis in the Goblint analyser and applied it to Linux kernel modules such as device drivers. One challenge in analyzing device drivers is how to model the rest of the kernel. Goblint uses a driver *harness* that assumes the worst possible interleavings of the device's file operations and interrupt handlers. Starting from the module initialization code, we track function pointers that are held in structs. Pointers passed to library functions are assumed to be potential call-backs and are analyzed as separate threads. These may interleave with each other as well as with the rest of the initialization code.

In the implementation, we also extended the basic approach to deal with nested static global data-structures such as structs containing arrays as well as well-behaved pointer arithmetic within structs. This is necessary for the analysis of the Linux API for doubly linked lists. This API provides macros which, e.g., calculate the start address of a struct from the address of a component. While these macros have a clean semantics, their implementation makes extensive use

File	Size (merged)	Time	Verified	Warnings
atmel_tclib	1317 lines	0,07 s	1	0
hwmon	1434 lines	0,23 s	1	0
enclosure	1510 lines	0,19 s	1	1
scsi_dh	4370 lines	0,57 s	2	0
dmaengine	4449 lines	0,83 s	3	0
scsi_rdac	4744 lines	0,81 s	1	0
usb_hcd	7340 lines	3,32 s	3	2
avc	7466 lines	1,68 s	2	1
ppp_generic	10818 lines	4,70 s	4	1

Table 1. Result of analysing kernel modules

of type casts, and addition and subtraction of pointers. Therefore, our implementation allows application of the address operator to arbitrary expressions evaluating to global addresses. Thus, pointers may no longer point to the beginnings of blocks. Moreover, a pointer variable whose value is obtained from the value of the pointer variable q by means of such kind of pointer arithmetic is put into the same region as q.

The results of running our analyzer on a number of different modules from the kernel is summarized in Table 1. We use the CIL analysis framework [19] as a front-end to parse and process these files. The sizes of the files in the table are the sizes of CIL's outputs after merging the modules with included headers and removing unused definitions. We ran these experiments on an Athlon 64 X2 3800+ machine under Kubuntu.[2]

For all these benchmarks, we are successful in automatically inferring the correlations between elements of lists and their corresponding locks and to verify that all accesses are protected. The numbers of shared variables for which we could verify a consistent locking scheme as well those for which conflicting accesses were found are listed in the table. The analyzer registers accesses to each element in a region separately; thus, if k linked lists have collapsed into a single region and there is a conflicting access through a pointer into this region, the number of warnings would be k and not one. The false alarms for these benchmakrs are mostly due to our imprecise harness. We will comment here only on two interesting benchmarks. The file `avc` is the access vector cache code of Security Enhanced Linux which served as the inspiration for the examples in this paper. The analyzer's output is the following:

```
Found correlation: avc_cache.latest_notif is guarded by
  lockset {notif_lock}
Found correlation: avc_cache.slots is guarded by
  lockset {avc_cache.slots_lock[*]}
Datarace over avc_callbacks:
  write in some thread with lockset: {} (avc.c:6953)
```

[2] The goblint website, `http://goblint.at.mt.ut.ee`, has detailed instructions on reproducing these benchmarks.

The asterisk in the second lockset is the analyzer's modest way of indicating that it has verified the correlation between the index expressions used when accessing list elements in the array of slots and the index expressions used to acquire a mutex from the array of locks. The analyzer warns about a "race" for `avc_callbacks`. While this is indeed a race in the context of this module alone, the function for registering callbacks are only used in the initialization code by the files using this module.

The file `dmaengine` is part of the hardware-neutral interface to the DMA subsystem. The programmers have commented in the source file: "The subsystem keeps two global lists, `dma_device_list` and `dma_client_list`. Both of these are protected by a mutex, `dma_list_mutex`." Our analyzer succeeds in verifying this.

7 Related work

Regions and ownership types have been used for compile-time garbage collection [27] or to ensure encapsulation in object-oriented languages [4]. More recently, analyzers have been developed for checking correct usage of region-based memory management APIs [1, 29]. Note, however, that the regions there need not be closed under reachability. For analyzing pointers, Gulwani and Tiwari [10] present a domain of quantified may- and must-equality pairs which can express similar invariants to ours. This analysis, while being extremely precise, has problems with dealing with doubly linked lists. Reachability in the presence of pointer arithmetic has been studied by Chatterjee et al. [3] who provide an annotation language for reasoning about the linked list API of Windows device drivers.

Precise abstractions of the heap have been provided by separation logic [22] and shape analysis [24]. Gopan et al. [8] present a shape analysis which allows reasoning about dynamic memory and the values of array elements, Gulwani et al. [9] present a *set cardinality analysis* which combines shape and numeric abstractions to reason about sizes of data-structures. Hackett and Rugina [11] present a shape analysis which is built on top of a partitioning of the heap into disjoint regions. These regions are derived from a standard points-to analysis and again not necessarily closed under reachability. Recent work has also provided methods for making shape analysis scale better [2, 15, 31] — at a certain loss in precision, e.g., by no longer tracking arrays.

Our main interest has been to provide efficient methods which are precise enough for analyzing data races in presence of dynamic data-structures and arrays. Rugina and Rinard [23] present techniques to avoid races by analyzing disjointness of accessed memory blocks. Naik and Aiken [18] propose *conditional must-not aliasing* to deal with locking schemes of various levels of granularity in Java. They introduce *disjoint reachability* analysis for dealing with medium-grained locking; however, their notion of disjointness is based on allocation sites, which is not helpful in cases such as Figure 2. We have experimented with some analyzers that perform race detection for C. We compared the following analyzers: Locksmith, a sound race detection tool based on type-based label-

Test	Goblint	Locksmith	Coverity	DDVerify
static	+/+	+/+	+/+	−/+
single list	+/+	+/−	−/+	
shared lists	+/+	+/−	−/+	
simple array	+/+	+/−	−/+	
shared array	+/+	+/−	−/+	

Table 2. Summary of comparison. For each idiom, "+" indicates success, while "–" indicates the existence of a False Negative / False Positive.

flow [21]; Coverity Prevent, a commercial bug-detection tool based on meta-compilation techniques [6]; and DDVerify, a device driver model-checker that checks for proper use of the kernel API [30].

We compared the tools on small test programs. For each test, there is a version with a race and one without races. The test *static* is the simplest possible race example, has a static global variable that should be protected by a static lock; *single list* contains a linked list where access to its nodes are protected by a single lock; *shared lists* has two lists that are protected by their own locks; still, there might be races due to sharing between elements in the lists; *simple array* contains an array of locks and an array of linked lists where the accesses should be properly correlated as in the examples of this paper; *shared array* is like the previous test, except there might be sharing between the linked lists of different array elements, hence there may a race although the correct lock is acquired. The summary of this comparison is shown in Table 2.

It seems that DDVerify checks other properties related to mutexes, e.g., double-acquisition of locks, but not whether accesses to globals are protected by the same locks. Locksmith and Coverity Prevent pass the first test, but already the simple linked list example is beyond their current capabilities. Locksmith complains on all tests, even when the program is perfectly safe; Coverity remains completely silent, even in the presence of races. Naturally, these analyzers have their advantages: Coverity checks a host of other properties, Locksmith deals with per-element locking, and DDVerify has an extremely precise automatic device driver harness mechanism; nevertheless, for medium-grained locking, Goblint is the clear winner.

8 Conclusion

We have presented a general approach to certify absence of data-races in C. In order to deal with dynamic data-structures, we provided a simple region analysis which allows to analyze reachability through field selection and dereferencing. We also indicated how this method can be extended to deal with arrays of regions and (well-behaved) pointer arithmetic. Our methods have been implemented in the efficient interprocedural data-race analyzer Goblint allowing us to verify locking schemes for dynamic data structures and arrays in the Linux kernel.

While we have analyzed benchmarks without modifying the original kernel code, in four of the benchmarks we only considered conflicts between write accesses. Read accesses are often protected by reader/writer locks, or more recently, the Read-Copy-Update mechanism. This poses a problem when the read accesses are protected at a coarser level of granularity than that of the write accesses. Thus, our failure to distinguish these would generate false alarms. Another challenge is to combine our technique here with methods dealing with per-element locking [20] in order to verify programs where some dynamically allocated structures, such as the per-device structure, contain linked lists and associated mutexes.

Acknowledgments. We thank Kalmer Apinis for assistance with the programming. Development of the analyzer is partially supported by the Estonian Science Foundation under grant no. 6713.

References

1. C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI'03*, pages 324–337. ACM Press, 2003.
2. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL'09*, pages 289–300. ACM Press, 2009.
3. S. Chatterjee, S. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing Low-Level software. In *TACAS'07*, LNCS, vol. 4424, pp. 19–33. Springer, 2007.
4. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA'98*, pages 48–64. ACM Press, 1998.
5. P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Recursive Programs. In E. Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 237–277. North-Holland Publishing Company, 1977.
6. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI'00*, pages 1–16. USENIX Association, 2000.
7. C. Fecht and H. Seidl. A Faster Solver for General Systems of Equations. *Sci. Comput. Programming*, 35(2):137–161, 1999.
8. D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *POPL'05*, pages 338–350. ACM Press, 2005.
9. S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *POPL'09*, pages 239–251. ACM Press, 2009.
10. S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV'07*, LNCS, vol. 4590, pp. 379–392. Springer, 2007.
11. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL'05*, pages 310–323. ACM Press, 2005.
12. M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Trans. Prog. Lang. Syst.*, 21(4):848–894, 1999.
13. M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6(2):133–151, 1976.

14. J. Knoop and B. Steffen. The Interprocedural Coincidence Theorem. In *CC'92*, LNCS, vol. 641, pp. 125–140. Springer, 1992.
15. R. Manevich, T. Lev-Ami, M. Sagiv, G. Ramalingam, and J. Berdine. Heap decomposition for concurrent shape analysis. In *SAS'08*, LNCS, vol. 5079, pp. 363–377, 2008.
16. M. Müller-Olm and H. Seidl. A note on Karr's algorithm. In *ICALP'04*, LNCS, vol. 3142, pp. 1016–1028. Springer, 2004.
17. M. Müller-Olm and H. Seidl. Upper adjoints for fast inter-procedural variable equalities. In *ESOP'08*, LNCS, vol. 4960, pp. 178–192. Springer, 2008.
18. M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL'07*, pages 327–338. ACM Press, 2007.
19. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: An infrastructure for C program analysis and transformation. In *CC'02*, LNCS, vol. 2304, pp. 213–228, 2002.
20. P. Pratikakis, J. S. Foster, and M. Hicks. Existential label flow inference via CFL reachability. In *SAS'06*, LNCS, vol. 4134, pp. 88–106. Springer, 2006.
21. P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-sensitive correlation analysis for detecting races. In *PLDI'06*, pages 320–331. ACM Press, 2006.
22. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE Press, 2002.
23. R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Prog. Lang. Syst.*, 27(2):185–235, 2005.
24. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Prog. Lang. Syst.*, 24(3):217–298, 2002.
25. H. Seidl, V. Vene, and M. Müller-Olm. Global invariants for analyzing multi-threaded applications. *Proc. of the Estonian Academy of Sciences: Phys., Math.*, 52(4):413–436, 2003.
26. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.
27. M. Tofte and L. Birkedal. A region inference algorithm. *ACM Trans. Prog. Lang. Syst.*, 20(4):724–767, 1998.
28. V. Vojdani and V. Vene. Goblint: Path-sensitive data race analysis. *Annales Univ. Sci. Budapest., Sect. Comp.*, 30:141–155, 2009.
29. X. Wang, Z. Xu, X. Liu, Z. Guo, X. Wang, and Z. Zhang. Conditional correlation analysis for safe region-based memory management. In *PLDI'08*, pages 45–55. ACM Press, 2008.
30. T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent linux device drivers. In *ASE'07*, pages 501–504. ACM Press, 2007.
31. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV'08*, LNCS, vol. 5123, pp. 385–398, 2008.