Static Data Race Analysis for C

Vesal Vojdani

Department of Computer Science, University of Tartu
J. Liivi 2, EE-50409 Tartu, Estonia
vesal@cs.ut.ee

Abstract. This is a survey of sound race detection techniques for C. While the basic lockset algorithm is at the core of most analyzers, the key challenge is to propagate pointer information context-sensitively. The bulk of the paper is, therefore, concerned with different solutions to the problem of context-sensitive pointer alias analysis in the context of race detection. These are the solutions used in the LOCKSMITH, CoBE, and RELAY analyzers. Other problems for sound static race detection are then discussed.

1 Introduction

A *multiple access data race* is a problem in low level concurrent programming where two threads simultaneously access the same shared memory location and at least one of the accesses is a write. Data races are known to kill innocent patients and leave unsuspecting citizens without electricity. As races are notoriously difficult to detect through testing, data race analysis has the potential to alleviate much suffering in the world. Unfortunately, race detection is also very difficult, but in the last couple of years some very impressive data race analyzers have been presented [6, 7, 10, 15].

The fundamental technique in static data races detection is to ensure that for each shared memory location, there exists (at least) one lock which is held whenever a thread accesses that memory location. Since at most one thread can hold a lock, this ensures mutually exclusive access to data and eliminates races of the kind specified above. In order to determine whether a common lock exists, one may compute the set of locks that are held by the executing thread for each program point. As execution may reach a given program point along many different paths, a sound analysis must conservatively track only locks that are held across all paths reaching that point. Having computed a set of definitely held locks, one only has to check the intersection of these locksets at each point where a given memory location is accessed. If the intersection is non-empty, one can conclude that there is no race on that given memory cell; otherwise, the analysis warns that there is a potential data race.

In order to apply this basic idea to analysing real C programs, one has to address the following challenges. First of all, determining statically the memory locations that are being accessed is not a trivial task. Even without dynamic memory allocation, pointers to static global variables need to be resolved. If two

```
int x;   mutex m1 = MUTEX_INIT;
int y,z; mutex m2 = MUTEX_INIT;

void munge(int *v, mutex *m) {
    lock(m); (*v)++; unlock(m); }

thread t1() {              thread t2() {
    munge(&x, &m1);            munge(&x, &m1);
    munge(&y, &m2);            munge(&y, &m1);
    munge(&z, &m2); }          munge(&z, &m2); }
```

Fig. 1. Example illustrating the need for context-sensitive pointer analysis [10].

distinct pointers, `p` and `q`, may alias, i.e., point to the same memory location, then syntactically distinct accesses, e.g., `p→data` and `q→data`, may partake in a data race. What makes this particularly challenging is that the locking and unlocking operations of C are not lexically scoped, so the information about pointers needs to be tracked *context-sensitively*, as will be illustrated through the following example.

Fig. 1 contains a simple program with two threads that execute calls to a `munge()` function. This function increments a shared variable while acquiring and releasing a mutex; both the variable and the mutex are given through pointer parameters. The effect of the function depends on the context in which it is called. If these calling contexts are conflated, the analysis will fail to deduce anything sensible about the program because `v` will be considered as pointing to any of the three shared variables, while `m` is considered as pointing to any of the two locks. As such functions commonly occur in real programs, context-sensitive propagation of pointer values is critical. Unfortunately, context-sensitive pointer information is very expensive to compute. The brute force approach to analyze the called function for each value of its parameters may not scale very well to large programs.

In this paper, we shall study three different solutions to this problem, solutions that are used in three different data race analyzers for C. In Section 2, we consider the type-based label flow used by Locksmith. In Section 3, we turn to bootstrapping to speed up pointer analysis in the CoBE framework. And in Section 4, we look at relative locksets that enable Relay to scale to millions of lines of code. Context-sensitivity is perhaps the main challenge for static race detection of C, but it is naturally not the only problem, as we will see in Section 5.

2 Type-Based Flow Analysis

The Locksmith analyzer [10] annotates the program with a type and effect system that compute everything needed for sound race detection. Type-based

program analysis is an interesting topic which is extensively studied in its own right. Formulating an analysis as a type system allows the application of many techniques from type theory to reason about properties of the analysis. For race detection, *polymorphism* allows the context-sensitive propagation of pointer information into functions, such as `munge` in Fig. 1, without needing to clone the body of the function [2]. Instead, a polymorphic function can be given a parametric type which is instantiated at each call site. As type systems are formulated for languages with higher-order functions, context-sensitive handling of function pointers comes for free.

The general approach of the LOCKSMITH analyzer is to accumulate access-lock correlation constraints. The analyzer collects for each access to a memory location ρ with the set of held locks L a *correlation constraint* $\rho \triangleright L$. Due to indirect accesses via pointers and parameter passing, other forms of constraints are also required, as will be explained below. Given a set of constraints C, the notation $C \vdash \rho \triangleright L$ indicates that the correlation $\rho \triangleright L$ can be derived from the constraints in C. The set $S(C, \rho) = \{L \mid C \vdash \rho \triangleright L\}$ denotes the set of all locksets that were held when accessing ρ. The location ρ is safely protected by a mutex, whenever the intersection of all locksets is non-empty: $\bigcap S(C, \rho) \neq \emptyset$. The accessed data is then said to be *consistently correlated* with the lockset. For the example program, the following correlation constraints are inferred:

`t1`:	$\rho_x \triangleright \{m_1\}$	$\rho_y \triangleright \{m_2\}$	$\rho_z \triangleright \{m_2\}$
`t2`:	$\rho_x \triangleright \{m_1\}$	$\rho_y \triangleright \{m_1\}$	$\rho_z \triangleright \{m_2\}$

The locations ρ_x and ρ_z are consistently correlated with the locks m_1 and m_2, respectively. As the intersection for ρ_y is empty, LOCKSMITH reports a race on the variable `y`. In order to obtain such information, one has to compute the set of held locks, generate all the constraints, and solve them.

As the focus of this survey is on the context-sensitive propagation of pointer information, we will first briefly discuss a few other features of the analyzer. The set of *definitely* held locks are computed flow-sensitively, meaning the control flow of the program is taken seriously; in contrast, pointer information is propagated flow-insensitively, meaning the analysis computes an over-approximation of all assignments within the body of a function independent of the order in which the assignments may be executed. Consider the following example:

```
void f() { int *p;
    p = &x; lock(&m1); *p = 41; unlock(&m1);
    p = &y; lock(&m2); *p = 42; unlock(&m2); }
```

LOCKSMITH will infer that `p` may point to either `x` or `y` all over this function. On the other hand, it computes the set of locks for each program point: when assigning 41 to `x` it is $\{m_1\}$, and when assigning 42 to `y` it is $\{m_2\}$. Due to flow-insensitivity, LOCKSMITH will infer the false constraints $\rho_x \triangleright \{m_2\}$ and $\rho_y \triangleright \{m_1\}$, which lead to a false alarm being reported. However, the coding style in this example, traversing the same pointer over different stack-allocated variables, is not that common, hence it is becoming increasingly popular in static analysis to ignore the flow within functions.

$$\textsc{Assign}\quad \frac{C;\Gamma \vdash e_1 : ref^{\rho}(\tau) \qquad C;\Gamma \vdash e_2 : \tau}{C;\Gamma \vdash e_1 := e_2 : \tau} \qquad \textsc{Seq}\quad \frac{C;\Gamma \vdash e_1 : \tau_1 \qquad C;\Gamma \vdash e_2 : \tau_2}{C;\Gamma \vdash e_1; e_2 : \tau_2}$$

$$\textsc{Sub}\quad \frac{C;\Gamma \vdash e : \tau_1 \qquad C \vdash \tau_1 \leq \tau_2}{C;\Gamma \vdash e : \tau_2} \qquad \textsc{Sub-Ref}\quad \frac{C \vdash \rho \leq \rho' \qquad C \vdash \tau \leq \tau' \qquad C \vdash \tau' \leq \tau}{C \vdash ref^{\rho}(\tau) \leq ref^{\rho'}(\tau')}$$

Fig. 2. Selection of monomorpic (intra-procedural) typing rules.

The flow-sensitive computation of the locksets is essentially achieved through a data flow analysis over the control flow graph of the program. In the type-based approach this amounts to using *state variables* to achieve flow-sensitive analysis. This allows the use of instantiation constraints for context-sensitive propagation of locksets, and there is additional cleverness with respect to function calls.

The flow-insensitive propagation of pointers within a function is achieved through *sub-typing* [11]. The idea is that each location has a type which associates it with a location label ρ; for example, the type of `&x` is $ref^{\rho_x}(int)$, a cell ρ_x containing an integer. Whenever there is a read or write to a variable of type $ref^{\rho}(\tau)$, one generates the constraint $\rho \triangleright L$ where L is the current lockset. The question is how to deal with indirect accesses through pointers.

Fig. 2 contains the relevant rules for intra-procedural pointer analysis. The rule for assignment states that a value of type τ can be stored into a memory cell of the same type $ref^{\rho}(\tau)$. Thus, in order to type the statements `p = &x; p = &y`, we need `p` to be of type $ref^{\rho_p}(\tau)$ where τ is equal to $ref^{\rho_x}(int)$ as well as $ref^{\rho_y}(int)$. This is only possible using the sub-typing rules, with which we can give the type $C;\Gamma \vdash \texttt{p} : ref^{\rho_p}(ref^{\rho_{xy}}(int))$ if $C \vdash \rho_x \leq \rho_{xy}$ and $C \vdash \rho_y \leq \rho_{xy}$ for a freshly generated location label ρ_{xy}. Inferring the type of the program thus requires that we *generate* such constraints. Then, these constraints are resolved using resolution rules such as the following:

$$C \cup \{\rho \leq \rho'\} \cup \{\rho' \triangleright L\} \quad \cup \Rightarrow \quad \{\rho \triangleright L\}$$

where $X \cup \Rightarrow Y$ is short-hand for $X \Rightarrow X \cup Y$. This rule propagates an access through a pointer to all its sub-types, so that from $\{\rho_{xy} \triangleright L,\ \rho_x \leq \rho_{xy},\ \rho_y \leq \rho_{xy}\}$, we also have $\rho_x \triangleright L$ and $\rho_y \triangleright L$.

We now add polymorphism to the type system in order to handle function calls context-sensitively. The traditional approach to polymorphic type inference universally quantifies all type variables that do not occur freely in the environment when a function is defined. Polymorphic types are then instantiated at each usage site by generating fresh variables and substituting in the type all occurrences of universally quantified types with the newly generated ones. In the context of constraint-based type inference, this involves the copying of the constraint sets, which can be quite large. Furthermore, copying the set of constraints for each call site would not constitute a significant gain over the brute

force approach of analyzing a separate copy of the function at each call. Instead, the flow of parameters into and out of a function can be captured as *instantiation constraints* $\tau_1 \preceq_p^i \tau_2$ where p is the polarity (direction of flow) and i is the unique identifier for each call-site. Note that this is a true instantiation in the sense that for each call site, there must exist a substitution ϕ_i such that $\phi_i(\tau_1) = \tau_2$; additionally, it expresses flow of information through the use of polarities.

It may be helpful to look at the two critical constraint resolution rules to understand the use of instantiation constraints.

$$C \cup \{\rho_1 \preceq_-^i \rho_0\} \cup \{\rho_1 \leq \rho_2\} \cup \{\rho_2 \preceq_+^i \rho_3\} \quad \cup\Rightarrow \quad \{\rho_0 \leq \rho_3\} \tag{1}$$

$$C \cup \{\rho \preceq_p^i \rho'\} \cup \{\rho \triangleright L\} \cup \{L \preceq^i L'\} \quad \cup\Rightarrow \quad \{\rho' \triangleright L'\} \tag{2}$$

The first rule propagates flow information in and out of a function context-sensitively, while the second propagates location and lock information into the function such that correlations constraints within the function are related to the values of parameters that went in. Locks do not need polarities because locks are unified as soon as there is flow between two lock labels. Polarities must take care of the flow in the presence of higher-order functions. Consider first the case where we have the following definitions:[1]

```
int *bar () { return &x; }
int  foo (int *(*fp)()) { return *fp(); }
```

When we now apply `foo(bar)`, we need to register that the location ρ_x has flowed into the function `foo` from the parameter `bar`. On the other hand, for the following definitions we have flow in the opposite direction:

```
int bar (int *p) { return *p; }
int foo (int (*fp)(int *)) { return fp(&x); }
```

Here, when we apply `foo(bar)`, the location ρ_x flows from `foo` into `bar`. We can summarize the two cases in a table and then generalize.

type of foo	type of bar	constraint
$\forall\rho.(void \to ref^{\rho}(int)) \to int$	$void \to ref^{\rho_x}(int)$	$\rho \preceq_-^i \rho_x$
$(ref^{\rho_x}(int) \to int) \to int$	$\forall\rho.ref^{\rho}(int) \to int$	$\rho_x \preceq_+^i \rho$

In general, the polarity of the argument to a function is flipped, denoted by $\bar{p}$. An instantiation constraint is generated for the function type at each call site and is then propagated according to the instantiation rule to handle polarities:

$$\textsc{Inst-Fun}\ \frac{C \vdash \tau_1 \preceq_{\bar{p}}^i \tau_2 \qquad C \vdash \tau_1' \preceq_p^i \tau_2'}{C \vdash \tau_1 \to \tau_1' \preceq_p^i \tau_2 \to \tau_2'} \qquad \textsc{Inst}\ \frac{C \vdash \tau \preceq_+ \tau'}{C; \Gamma, f : \forall\vec{l}.\tau \vdash f^i : \tau'}$$

[1] Definitions in C are read from the inside out; `()` has a higher priority than `*`. The parameter `int *(*fp)()` is a pointer to – a function that returns – a pointer to – an integer.

The rule for instantiation is simplified to ignore the free variables that could not be universally quantified at the let-binding.

We now return to the motivating example of Fig. 1. The type of the `munge()` function is $C; \Gamma \vdash \forall \rho_v, m \,.\, ref^{\rho_v}(int) \times lock(m) \to void$ where $C \vdash \rho_v \triangleright \{m\}$. We consider the instantiation constraints generated in order to type thread one:

$$ref^{\rho_v}(int) \times lock(m) \to void \preceq^1_+ ref^{\rho_x}(int) \times lock(m_1) \to void$$
$$ref^{\rho_v}(int) \times lock(m) \to void \preceq^2_+ ref^{\rho_y}(int) \times lock(m_2) \to void$$
$$ref^{\rho_v}(int) \times lock(m) \to void \preceq^3_+ ref^{\rho_z}(int) \times lock(m_2) \to void$$

We resolve the first one to $ref^{\rho_v}(int) \times lock(m) \preceq^1_- ref^{\rho_x}(int) \times lock(m_1)$ which (by obvious rules I have not shown) simplifies to $\{\rho_v \preceq^1_- \rho_x,\ m \preceq^1 m_1\}$. When this is conjoined with the correlation constraint $\rho_v \triangleright \{m\}$, the constraint resolution rule (2) allows us to infer $C \vdash \rho_x \triangleright \{m_1\}$. Analogously, we obtain all the other constraints required to check for races.

This is *almost* the whole story; a very prominent feature of the type system has been deliberately ignored in this presentation and eradicated from the typing rules. LOCKSMITH uses an effect system to enforce linearity among locks. This is required for sound analysis because a dynamically allocated lock might be re-allocated:

```
mutex *m = malloc(); mutex_init(m);
lock(m); x++; unlock(m);
m = malloc(); mutex_init(m);
lock(m); x++; unlock(m);
```

Here, the lock pointer m is not referring to the same lock. A sound analyzer should consider this, even if it rarely occurs in real programs. In fact, the LOCKSMITH authors turn it off when analyzing real programs because the analysis does not scale with this feature turned on.

3 Bootstrapping

Kahlon et. al [6] present a technique for fast must-alias analysis of lock pointers and a shared variable discovery algorithm. Unfortunately, it is unclear how the ideas presented in the paper apply to the example of Fig 1, where may-aliasing of pointers to shared variables and must-aliasing of lock pointers are to be jointly propagated context-sensitively.

The general approach is to first identify shared variables and the location where these shared variables are accessed. If the same shared variable can be accessed by two different threads simultaneously and the set of locks they hold are disjoint, a race warning is emitted. The second step is, therefore, to identify the set of held locks. Here, a sophisticated must-alias analysis is proposed based on bootstrapping and procedure summarization. Finally, warning reduction techniques are applied.

The suggested method for shared variable discovery is somewhat puzzling. The idea seems to be that one should conservatively consider all global variables and pointers passed to external functions as shared. Thus much makes sense, but in order to deal with local aliases to shared data, aliased pointers are also considered as shared (with the minor refinement that only those pointers are added which are instrumental in resulting in a true access rather than just propagating address information.) This is surprising because when there is an indirect access to a shared variable through a pointer, we would expect an attempt to resolve the pointer, rather than register the access with the pointer and ensure that accesses to the pointer are safe.

The must-alias analysis of pointers is based on the idea of Bootstrapping alias analyses [3]. This is an approach marketed by Vineet Kahlon to "leverage the combination of *divide and conquer*, *parallelization* and *function summarization*." The key idea is to use a succession of alias analyses of increasing precision such that the rough partitioning of the first alias analysis allows the more precise ones to run on a much smaller problem instances. This only works if one can prove that the equivalence classes computed by the more coarse-grained analyses and the slices of programs that one considers for each cluster suffice to correctly compute the refined alias information at the next stage.

One suitable pointer analysis to begin the bootstrapping process is Steensgaard's alias analysis. This is nearly identical to the flow-insensitive pointer analysis from Section 2, except instead of introducing sub-typing when a pointer `p` may refer to two distinct locations, the locations are unified, i.e., considered as a single abstract location. This results in a partitioning of pointers into equivalence classes. Practically, the partitioning means that one only need to deal with a single equivalence class at a time, and for locks this typically involves 2–3 pointers.

Having clustered the set of pointers, more expensive must-alias analysis can be applied. The analysis proposed in the article is based on *Maximally Complete Update Sequences* which can be used to characterize aliasing. These can then be used to compute a *procedure summary*, i.e., a finite representation of the abstract effect of applying that function. A summary is computed once and for all and then applied whenever the function is called, achieving context-sensitivity without the cost of cloning, as was previously achieved through polymorphism. We begin with the notion of update sequences which are central to this approach. The goal is to characterize must-aliasing in terms of chains of assignment: two pointers p and q must alias precisely when there exists some location a and chains of assignments π_1 and π_2 that are semantically equivalent to $p = a$ and $q = a$. This idea is formalized as follows.

Definition 1 (Complete Update Sequence [3]). *Let $\lambda\colon l_0, \ldots, l_m$ be a sequence of successive program locations and let π be the following sequence of pointer assignments along λ:*

$$l_{i_1} : p_1 = a_0; \qquad l_{i_2} : p_2 = a_1; \qquad \ldots \qquad l_{i_k} : p_k = a_{k-1};$$

Then π is called a complete update sequence from p to q leading from locations l_0 to l_m if the following conditions hold:

- *a_0 and p_k are semantically equivalent (i.e., evaluate to the same location) to p and q at locations l_0 and l_m, respectively.*
- *for each j, a_j is semantically equivalent to p_j at l_{i_j},*
- *for each j, there does not exist any (semantic) assignment to pointer a_j between locations l_{i_j} and $l_{i_{j+1}}$; to a_0 between l_0 and l_{i_1}; and to p_k between l_{i_k} and l_m along λ.*

Thus, a complete update sequence from p to q (leading from l_0 to l_m) means that executing the code snippet between l_0 and l_m has an effect on q which is equivalent to performing the assignment $q = p$ at location l_m.

Kahlon gives the example shown in Fig. 3 to illustrate the idea. We can see that line 4 by itself is a complete update sequence from b to a (leading from 1 to 4) because executing the snippet will result in assigning b to a though the indirect assignment on line 3. Note that the sequence is effectively equivalent to $a = b$ being performed at the end, but the single assignment does not adequately capture the effect of these lines on the variable a because b has obtained its value from c in the first line, which we have ignored. In contrast, the update sequence $1, 4$ is also a complete update sequence leading from 1 to 4, but it is from c to a. This update sequence really captures what happens to the pointer a when executing lines 1 to 4. Such update sequences are called *maximally complete*.

```
int main() {
    int *a, *b, *c;
    int **x, **y;
    1: b = c;
    2: x = &a;
    3: y = &b;
    4: *x = b; }
```

Fig. 3. Update Sequence

Formally, the maximally complete update sequence for a pointer q leading from location l_0 to l_m along λ is the complete update sequence π of maximum length, over all pointers p, from p to q (leading from locations l_0 to l_m) occurring along λ. We can now characterize aliasing as follows: pointers p and q must alias at program point l if and only if there exists a pointer a with maximally complete update sequences to both p and q. Since the goal is to obtain an efficient summary of a procedure's effect on aliasing, the summaries track maximally complete update sequences.

The summary of a function f is the set of triples of the form (p, l, A), where p is the pointer of interest, l is an (important) program point, and A is the set of all pointers q such that there is a complete update sequence from q to p along every path leading from the entry of the function to the program point l. This summary is computed through a process that is reminiscent of weakest pre-condition computation: we start with the summary of $(p, l, \{p\})$ and work backwards in the control flow graph so that an assignment $p = q$ has the effect of replacing p with q in the set A giving us $(p, l, \{q\})$. When we reach the entry point of the function we have computed the summary for the aliases of p at location l. The effect of applying a function on the pointer p is the summary for that pointer at the exit location. Thus, when the analysis needs to consider the

effect of a function call, it looks up the summary for each pointer in A. It would be interesting to apply the algorithm to the example in Fig. 3, but unfortunately the algorithm, as described in the paper, does not consider the case of indirect updates. (The example is not from the same paper.)

A more serious problem is that this approach does not handle the running example from the introduction: although pointers are updated context-sensitively at each call site, the different calling contexts are not distinguished within the body of the function. The may-aliasing of shared variables are not really discussed in the paper. In a more recent paper, Kahlon et al. [5] propose a Context Sensitive Call-Graph construction, but in order to deal with our motivating example, the must- and may-alias information need to be propagated *into* functions to infer correlations context-sensitively. In the next section, we will study a summarization approach which achieves this goal by summarizing locksets together with *guarded accesses*.

4 Relative Locksets

The RELAY analyzer [15] provides a very simple and elegant solution to the problem of context-sensitive pointer analysis. It relies on the concept of a *relative lockset* to describe the changes in the locksets relative to the function entry point. It also accumulates accessed memory locations relative to the entry point, thus obtaining a set of guarded accesses which are expressed in relation to the entry parameters of the function. This summarizes the effect of the function, which is then used at call sites by plugging in the values of the parameters at any given calling context. Consider the following function.

```
void foo(struct node *x) {
    lock(x→mtx1); x→f = 7; unlock(x→mtx2);
}
```

Its summary would consist of two components: first, the relative lockset at the end of the call, which is to add `x→mtx1` and remove `x→mtx2` from the set of mutexes of the caller; and second, a list of relative accesses, which in this case is just the access to `x→f` with the relative lockset of adding `x→mtx1` to set of mutexes of the caller.

Functions are processed bottom-up in the call graph. Any function whose callees have been summarized can be analyzed in separation; this allows parallelization of the analysis. For each function, three analyses are performed: symbolic execution, relative lockset analysis, and guarded access analysis. The foundation for the other analyses is laid by the symbolic execution which aims to express the values of program variables in terms of the "incoming" values of the function's parameters. The analysis tracks for each program point a symbolic map $\Sigma : \mathbb{O} \to \mathbb{V}$ from symbolic L-values to symbolic R-values defined as follows.

$$o ::= x \mid p \mid o.f \mid *o \qquad \text{L-values}$$
$$v ::= \bot \mid \top \mid i \mid \mathit{init}(o) \mid \mathit{may}\{os\} \qquad \text{R-values}$$

R-Values include $\bot$ (unassigned), $\top$ (unknown), integers, the incoming (initial) value of some L-Value, and a may-points-to set of L-Values. For the symbolic L-Values, the meta-variable x ranges over program variables, while p ranges over representative summary nodes computed by an external flow-insensitive (hence very fast) points-to analysis. This is primarily to ensure termination. Consider the following example:

```
void foo(struct node *x) {
    struct list *y = x→first;
    while (y) { y→data = 5; y = y→next; }
}
```

Before executing the loop the mapping is $[\texttt{x} \mapsto \mathit{init}(\texttt{x}),\ \texttt{y} \mapsto \mathit{may}\{\texttt{x}{\rightarrow}\texttt{first}\}]$, where we use $o{\rightarrow}f$ as a synonym for $(*o).f$. After executing the loop, the pointer p may now point to a possibly infinite set $\{\texttt{x}{\rightarrow}\texttt{first}, \texttt{x}{\rightarrow}\texttt{first}{\rightarrow}\texttt{next}, \texttt{x}{\rightarrow}\texttt{first}{\rightarrow}\texttt{next}{\rightarrow}\texttt{next}, \ldots\}$, which one may want to simply replace with a single summary node. The main idea of relative locksets is independent of the precise symbolic execution analysis.

The second step is the computation of relative locksets using the information from the symbolic execution to evaluate lock expressions. A relative lockset L is the pair (L_+, L_-) of definitely acquired and possibly released locks since the beginning of the function. The ordering is as expected with a must- and may-set:

$$(L_+, L_-) \sqsubseteq (L'_+, L'_-) \iff L_+ \supseteq L'_+ \text{ and } L_- \subseteq L'_-$$

The relative lockset at the exit of a function f is the summary lockset L_f which is used whenever the function is invoked. The analysis treats everything as function calls with `lock(&l)` and `unlock(&l)` being simulated as functions with summaries $(\{l\}, \emptyset)$ and $(\emptyset, \{l\})$, respectively. Summaries are applied according to the following scheme: rebind the formals in L_f to the values of the arguments computed by the symbolic execution and update the lockset before the call L with the effect of the summary L_f. The effect of a call $f(a)$, where a is the argument list and p is the list of formal parameters of f, is computed as $\mathit{update}(L, L_f[p \mapsto \mathit{eval}(a)])$ where

$$\mathit{update}((L_+, L_-), (L'_+, L'_-)) = ((L_+ \cup L'_+) \setminus L'_-,\ (L_- \cup L'_-) \setminus L'_+)$$

This updates the lockset by adding the effect of the summary and then removing any lock that may have been released from the set of definitely held locks as well as removing any lock that the called function definitely ends up holding from the set of released locks.

Finally, the set of guarded accesses are computed for each function. A guarded access is a triple $a = (o, L, k)$, where $o \in \mathbb{O}$ is an L-value being accessed, L is the relative lockset and k indicates whether the access was a read or a write. The propagation of guarded accesses by using summaries is very similar to the relative locksets, although computing the set of accesses does not need to be flow-sensitive, but can simply traverse assignments in any order.

Let us return to the example program from the introduction. The guarded access for `munge` is $\{(*\mathtt{v},\ (\{\mathtt{m}\},\emptyset), \text{write})\}$ which at each instantiation is rebound to the arguments and the lockset is updated with the relative lockset of the caller, which in this case is empty. Instantiating the arguments requires resolving the may-points-to sets: $*\mathtt{v}[\mathtt{v} \mapsto \mathit{eval}(\&\mathtt{x}) = \mathit{may}\{\mathtt{x}\}] = \mathtt{x}$. Here, we had a singleton points-to set, but in principle a guarded access of a function may need to be instantiated to multiple accesses if the caller gave an ambiguous pointer as parameter. In our simple example, we obtain for `t1` and `t2`:

$$\mathtt{t1}: \quad \{(\mathtt{x},\ (\{\mathtt{m1}\},\emptyset),\ \text{write}),\ (\mathtt{y},\ (\{\mathtt{m2}\},\emptyset),\ \text{write}),\ (\mathtt{z},\ (\{\mathtt{m2}\},\emptyset),\ \text{write})\}$$
$$\mathtt{t2}: \quad \{(\mathtt{x},\ (\{\mathtt{m1}\},\emptyset),\ \text{write}),\ (\mathtt{y},\ (\{\mathtt{m1}\},\emptyset),\ \text{write}),\ (\mathtt{z},\ (\{\mathtt{m2}\},\emptyset),\ \text{write})\}$$

Race warnings are generated by considering pairs of thread entry points (here there is only one such pair: `t1` and `t2`) and identifying whether there exists a pair of accesses a_1 and a_2 that conflict, i.e., their L-values alias, but the locksets are disjoint (and at least one is a write). In the example, the accesses $\{(\mathtt{y},\ (\{\mathtt{m2}\},\emptyset),\ \text{write})$ and $\{(\mathtt{y},\ (\{\mathtt{m1}\},\emptyset),\ \text{write})$ conflict, hence the correct warning is flagged.

5 Remaining challenges for static race detection

This paper has focused on the main challenge of static race detection for C: making context-sensitive alias analysis scale. In this section, we will briefly consider other issues in race detection. First, we face the fact that mutexes are not the only means to ensure mutually exclusive access to data. Second, we consider the additional challenges to pointer analysis posed by dynamic memory allocation. Finally, we consider the challenges of dealing with slightly different locking operation, e.g., semaphores and reader-writer locks, as well as cases when control flow depends on the acquired locks. These problems are based on the classification of false alarms by the authors of RELAY.

Synchronization-Sensitivity. The safest way to ensure freedom from races is to only run a single thread. Even in a multi-threaded programs, a thread may not be running in parallel with all other threads at all times. There are many mechanisms to achieve lock-free synchronization. These are often very hard to analyze, as the following example will illustrate, but something can be done by attempting to track thread identities and inferring which threads may possibly run in parallel.

Consider, as an example [10], the scenario where we have a main thread with k worker threads. The main thread maintains an array A with k elements, one for each thread, such that $A[i]$ is manipulated by thread i. Furthermore, suppose the main thread initializes the array before spawning worker threads and processes the array after all workers have terminated. Although there is no locking, the program is free from races because the main thread may only access the array when the workers do not and the workers follow a convention that ensures mutually exclusive access.

There are typically distinct *temporal* phases in a program, such as initialization, processing, and post-processing. An analyzer must, therefore, determine not only which threads may run in parallel, but whether two given accesses may actually conflict, taking into account which threads accesses what data at what time. In the example, the main is still running when the worker threads start, but it no longer touches the array. The conventional approach is to attempt to partially order statements when it is clear that something must *happen before* another operation. A race can then only occur on two accesses that lack ordering constraints. This is exploited by many dynamic analyzers, such as the Intel Thread Checker. For static analysis of C, acquisition histories, proposed by Kahlon et al. [4], is an interesting approach to improving synchronization-sensitivity.

Although some progress has been made in this regard [5], much of the practical synchronization in programs are based on deeper properties of the program logic, or rely on various spooky synchronization primitives, like signals, conditional variables, wait-queues, etc.

Races in the Heap. It should come as no surprise that *dynamic* memory allocation is extremely difficult for *static* analyzers. Since many serious program errors relate to memory safety, the analysis of the heap is currently a highly active area of research. When it comes to race detection tools, most of them still rely on summarizing all data allocated at a given program point into a single representative "blob". This can be highly imprecise: the alias analysis used by RELAY had in the extreme case merged over 10,000 objects into a single blob.

The problem with summarization for race detection is that on the one hand, we have to treat an accesses to a blob as an access to *all* the objects it represents, while on the other hand, we must consider the locking of a blob as taking *none* of the locks. This asymmetry is a consequence of having to ensure that if two thread *may* access the same element, they *must* lock the same lock. This problem can be illustrated even without dynamic memory. It is quite natural for an object to contain a dedicated lock which ensures mutually exclusive access to its data fields, as in the following example:

```
struct { int datum; mutex mtx; } A, B;
if (*) p = &A; else p = &B;
lock(p→mtx); p→datum++; unlock(p→mtx);
```

After the non-deterministic branching the pointer `p` may point to either A or B, so when we acquire the lock `p→mtx` we may hold either one of them, and thus neither of them is *definitely* held. Nevertheless, it is obvious that the above code is correct; after all, the same pointer is used for both the access and for the locking. The situation is precisely the same when `p` points to a blob of dynamically allocated memory.

The LOCKSMITH analyzer can deal with the per-element locking just described by means of existential quantification in their label flow [9]. This requires programmer annotations to explicitly introduce quantification whenever an element of a blob should be treated as a concrete representative for elements in

the blob. In order to avoid annotations, one can use pointer must-equality analysis to reason about the access and the lock relative to the root pointer (e.g., `p→datum` is accessed while holding `p→mtx`) and instantiate the found invariant to all elements that `p` may point to, including blobs [13].

There are naturally many complicated locking schemes. Naik and Aiken [8] discuss different granularities of locking schemes from coarse-grained to fine-grained. They propose disjoint reachability analysis to deal with medium-grained locking, e.g., one lock for a linked list. The technique, which is based on conditional aliasing for Java, is hard to transfer to the C scenario where the lock for a linked list may not be contained in the head object, making conditional aliasing unhelpful. If the locks can be statically named, *region analysis* can be used to verify medium-grained locking schemes such as synchronized hash-tables [12].

There is an additional concern with dynamically allocated memory which reminds of the problems of synchronization sensitivity. A dynamically allocated object often has its own life cycle, such as initialization, consumption, and destruction. Although we consider a dynamically allocated object part of the global heap, a freshly allocated object is only accessible by the thread that allocated it until the thread *publicizes* the element by connecting it to the rest of the heap, e.g., adding it to a linked list. Similarly, near the end of an objects life-time, it may be *privatized* by being removed from the data structure where it lived, so that it is only accessible by a single thread.

Discovering when a freshly allocated object becomes shared is much easier than dealing with privatization. For example, LOCKSMITH performs an effect analysis to discover when allocated variables escape the thread. Once an object has escaped and is part of the heap, one can only discover its removal through careful analysis of the heap. This requires the use of scalable concurrent shape analysis of low-level C, which in spite of much recent progress is far from reality.

Conditional locking and variations on locking. The authors of RELAY write, "several false warnings generated by RELAY were because the program checks some condition to determine whether to acquire locks, and later, checks a correlated condition to determine whether the access should occur. Unfortunately, the acquisition of the lock and the actual access occur in different blocks or functions thereby introducing a path-sensitivity problem." Path-sensitivity is the ability of the analysis to distinguish feasible paths from infeasible one:

```
if (do_work) pthread_mutext_lock(&m);
if (do_work) work++;
if (do_work) pthread_mutext_unlock(&m);
```

There are 8 paths in the above program, but only two are valid paths. Path-sensitivity is also required in situations where thread creation and locking operations may fail, such as pthread's `trylock` and the kernel's `lock_interruptible`. The return value is then used to correlate the different locksets that result from potentially failing locking operation with the values of program variables. One method for achieving path-sensitivity is called *property-simulation* [1], which analyzes the state of the program for each configuration of the relevant property,

namely locksets. This approach is used by the Goblint analyzer to deal with conditional locking and possibly failing locks [14].

When analyzing the Kernel, especially older versions, some synchronization is achieved through the use of semaphores. These allow nested locking and unlocking and count the number of times they have been locked so that the lock is only released when all acquisitions are matched with a release. Naturally, one can approximate semaphores as mutexes, counting the first release as a release of the semaphore. This leads to false alarms; however, unless semaphores are lexically scoped, semaphore locking schemes are probably undecidable.

The kernel also uses reader-writer (R/W) locks and the read-commit-update (RCU) mechanism to synchronize linked lists. R/W-locks allow multiple readers to acquire a lock, as long as nobody has not acquired for writing, in which case only a single thread can hold the lock. R/W-locks can be dealt with fairly easily by tracking whether a lock was acquired for reading or writing and only considering a reader lock to guard an access if it is a read access: write accesses while holding a reader lock are just as bad as having no lock at all. The kernel's RCU mechanism is usually defined in terms of R/W-locks, but the actual implementation is far more efficient. Although these should be simple to handle in principle, race detection tools tend to not deal with these variations on locking mechanisms.

Acknowledgements. I'm grateful to the anonymous referees for pointing out serious problems in the submitted version of the paper.

References

1. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. In: PLDI'02. pp. 57–68. ACM Press (2002)
2. Fähndrich, M., Rehof, J., Das, M.: Scalable context-sensitive flow analysis using instantiation constraints. In: PLDI'00. pp. 253–263. ACM Press (2000)
3. Kahlon, V.: Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In: PLDI'08. pp. 249–259. ACM Press (2008)
4. Kahlon, V., Ivančić, F., Gupta, A.: Reasoning about threads communicating via locks. In: CAV'05. LNCS, vol. 3576, pp. 505–518. Springer (2005)
5. Kahlon, V., Sinha, N., Kruus, E., Zhang, Y.: Static data race detection for concurrent programs with asynchronous calls. In: ESEC/FSE'09. pp. 13–22. ACM Press (2009)
6. Kahlon, V., Yang, Y., Sankaranarayanan, S., Gupta, A.: Fast and accurate static data-race detection for concurrent programs. In: CAV'07. LNCS, vol. 4590, pp. 226–239. Springer (2007)
7. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: PLDI'06. pp. 308–319. ACM Press (2006)
8. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL'07. pp. 327–338. ACM Press (2007)
9. Pratikakis, P., Foster, J.S., Hicks, M.: Existential label flow inference via CFL reachability. In: SAS'06. LNCS, vol. 4134, pp. 88–106. Springer (2006)

10. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Context-sensitive correlation analysis for detecting races. In: PLDI'06. pp. 320–331. ACM Press (2006)
11. Rehof, J., Fähndrich, M.: Type-based flow analysis: from polymorphic subtyping to CFL-reachability. In: POPL'01. pp. 54–66. ACM Press (2001)
12. Seidl, H., Vojdani, V.: Region analysis for race detection. In: SAS'09. LNCS, vol. 5673, pp. 171–187. Springer (2009)
13. Seidl, H., Vojdani, V., Vene, V.: A smooth combination of linear and Herbrand equalities for polynomial time must-alias analysis. In: FM'09. LNCS, vol. 5850, pp. 644–659. Springer (2009)
14. Vojdani, V., Vene, V.: Goblint: Path-sensitive data race analysis. Annales Univ. Sci. Budapest., Sect. Comp. 30, 141–155 (2009)
15. Voung, J.W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: ESEC/FSE'07. pp. 205–214. ACM Press (2007)