LOCKSMITH: Practical Static Race Detection for C

Polyvios Pratikakis
University of Maryland
and
Jeffrey S. Foster
University of Maryland
and
Michael Hicks
University of Maryland

LOCKSMITH is a static analysis tool for automatically detecting data races in C programs. In this paper, we describe each of LOCKSMITH's component analyses precisely, and present systematic measurements that isolate interesting tradeoffs between precision and efficiency in each analysis. Using a benchmark suite comprising standalone applications and Linux device drivers totaling more than 200,000 lines of code, we found that a simple no-worklist strategy yielded the most efficient interprocedural dataflow analysis; that our sharing analysis was able to determine that most locations are thread-local, and therefore need not be protected by locks; that modeling C structs and void pointers precisely is key to both precision and efficiency; and that context sensitivity yields a much more precise analysis, though with decreased scalability. Put together, our results illuminate some of the key engineering challenges in building LOCKSMITH and data race detection analyses in particular, and constraint-based program analyses in general.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

General Terms: Data Race, Race Detection, Static Analysis
Additional Key Words and Phrases: context-sensitive, correlation inference, sharing analysis, contextual effects, Locksmith

1. INTRODUCTION

Multithreaded programming is becoming increasingly important as parallel machines become more widespread. Dual-core processors are fairly common, even among desktop users, and hardware manufacturers have announced prototype chips with as many as 80 [intel.com 2007] or 96 [news.com 2007] cores. It seems inevitable that to take advantage of these resources, multithreaded programming will become the norm even for the average programmer.

However, writing multithreaded software is currently quite difficult, because the programmer must reason about the myriad of possible thread interactions and may need to consider unintuitive memory models [Manson et al. 2005]. One particularly important problem is *data races*, which occur when one thread accesses a memory location at the same time another thread writes to it [Lamport 1978]. Races make the program behavior unpredictable, sometimes with disastrous consequences [Leveson and Turner 1993; Poulsen 2004]. Moreover, race-freedom is an important property in its own right, because race-free programs are easier to un-

derstand, analyze and transform [Alexandrescu et al. 2005; Reynolds 2004]. For example, race freedom simplifies reasoning about code that uses locks to achieve atomicity [Flanagan and Freund 2004; Flanagan and Qadeer 2003].

In prior work, we introduced a static analysis tool called LOCKSMITH for automatically finding all data races in a C program [Pratikakis et al. 2006b]. LOCKSMITH aims to soundly detect data races, and works by enforcing one of the most common techniques for race prevention: for every shared memory location ρ, there must be some lock ℓ that is held whenever ρ is accessed. When this property holds, we say that ρ and ℓ are *consistently correlated*. In our prior work, we formalized LOCKSMITH by presenting an algorithm for checking consistent correlation in $\lambda_{\triangleright}$, a small extension to the lambda calculus that models locks and shared locations. We then briefly sketched some of the necessary extensions to handle the full C language, and described empirical measurements of LOCKSMITH on a benchmark suite.

In this paper, we discuss in detail the engineering aspects of scaling the basic algorithms for race detection to the full C language. We present our algorithms precisely on a core language similar to $\lambda_{\triangleright}$, which captures the interesting and relevant features of C with POSIX threads and mutexes. We subsequently extend it with additional features of C, and describe how we handle them in LOCKSMITH. We then perform a systematic exploration of the tradeoffs between precision and efficiency in the analysis algorithms used in LOCKSMITH, both in terms of the algorithm itself, and in terms of its effects on LOCKSMITH as a whole. We performed measurements on a range of benchmarks, including C applications that use POSIX threads and Linux kernel device drivers. Across more than 200,000 lines of code, we found many data races, including ones that cause potential crashes. Put together, our results illuminate some of the key engineering challenges in building LOCKSMITH in particular, and constraint-based program analyses in general. We discovered interesting—and sometimes unexpected—conclusions about the configuration of analyses that lead to the best precision with the best performance. We believe that our findings will prove valuable for other static analysis designers.

We found that using efficient techniques in our dataflow analysis engine eliminates the need for complicated worklist algorithms that are traditionally used in similar settings. We found that our sharing analysis was effective, determining that the overwhelming majority of locations are thread-local, and therefore accesses to them need not be protected by locks. We also found that simple techniques based on scoping and an intraprocedural uniqueness analysis improve noticeably on our basic thread sharing analysis. We discovered that field sensitivity is essential to model C structs precisely, and that distinguishing each type that a void $*$ may represent is key to good precision. Lastly, we found that context sensitivity, which we achieve with parametric polymorphism, greatly improves LOCKSMITH's precision.

The next section presents an overview of LOCKSMITH and its constituent algorithms, and serves as a road map for the rest of the paper.

2. OVERVIEW

Fig. 1 shows the architecture of LOCKSMITH, which is structured as a series of subanalyses that each generate and solve constraints. In this figure, plain boxes represent processes and shaded boxes represent data. LOCKSMITH is implemented using

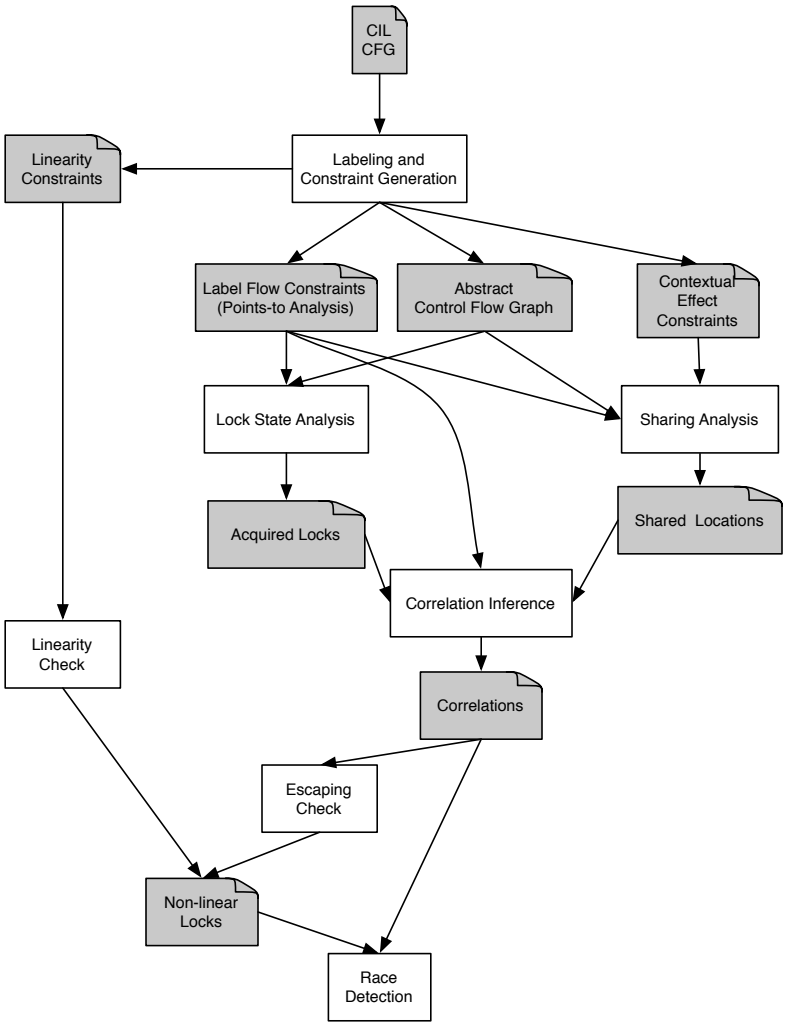


Fig. 1. LOCKSMITH architecture

CIL, which parses the input C program and simplifies it to a core sub-language [Necula et al. 2002]. As of now, LOCKSMITH supports the core of the POSIX and Linux kernel thread API, namely the API calls for creating a new thread, the calls for allocating, acquiring, releasing and destroying a lock, as well as `trylock()`. Currently, we do not differentiate between read and write locks.

In the remainder of this section, we sketch each of LOCKSMITH's components and then summarize the results of applying LOCKSMITH to a benchmark suite. In the subsequent discussion, we will use the code in Fig. 2 as a running example.

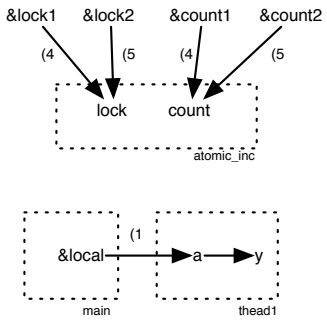
The program in Fig. 2 begins by defining four global variables, locks lock1 and lock2 and integers count1 and count2. Then lines 4–9 define a function atomic_inc that takes pointers to a lock and an integer as arguments, and then increments

```
1   pthread_mutex_t lock1, lock2;
2   int count1 = 0, count2 = 0;
3
4   void atomic_inc(pthread_mutex_t *lock,
5           int *count) {
6     pthread_mutex_lock(lock);
7     *count++;
8     pthread_mutex_unlock(lock);
9   }
10
11  int main(void) {
12    pthread t1, t2, t3;
13    int local = 0;
14
15    pthread_mutex_init(&lock1, NULL);
16    pthread_mutex_init(&lock2, NULL);
17
18    local++;
19    pthread_create1(&t1, NULL, thread1, &local);
20    pthread_create2(&t2, NULL, thread2, NULL);
21    pthread_create3(&t3, NULL, thread3, NULL);
22  }
23  void *thread1(void *a) {
24    int *y = (int *) a; /* int* always */
25    while(1) {
26      *y++; /* thread-local */
27    }
28  }
29
30  void *thread2(void *c) {
31    while(1) {
32      pthread_mutex_lock(&lock1);
33      count1++;
34      pthread_mutex_unlock(&lock1);
35      count2++; /* access without lock */
36    }
37  }
38
39  void *thread3(void *b) {
40    while(1) {
41      /* needs polymorphism for atomic_inc */
42      atomic_inc4(&lock1, &count1);
43      atomic_inc5(&lock2, &count2);
44    }
45  }
```

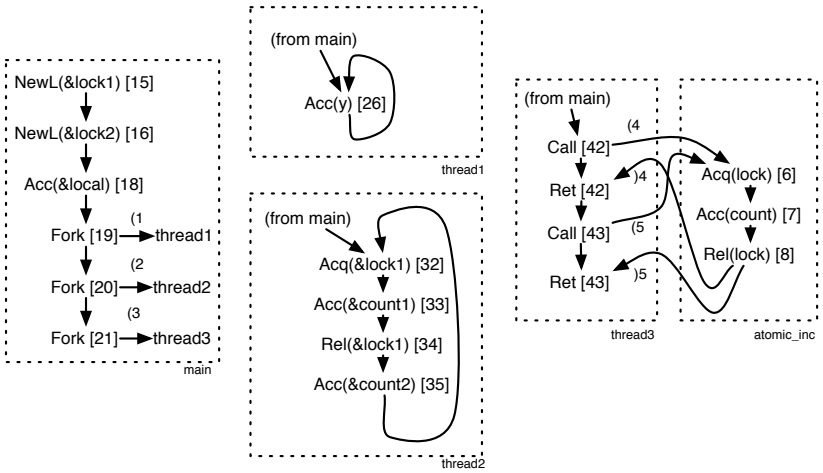
Fig. 2. Example multithreaded C program

the integer while holding the lock. The main function on lines 11–22 allocates an integer variable local, initializes the two locks, and then spawns three threads that execute functions thread1, thread2 and thread3, passing variable local to thread1 and NULL to thread2 and thread3. We annotate each thread creation and function call site, except calls to the special mutex initialization function, with a unique index i, whose use will be explained below. The thread executing thread1 (lines 23–28) first extracts the pointer-to-integer argument into variable y and then continuously increments the integer. The thread executing thread2 (lines 30–37) consists of an infinite loop that increases count1 while holding lock lock1 and count2 without holding a lock. The thread executing thread3 (lines 39–45) consists of an infinite loop that calls atomic_inc twice, to increment count1 under lock1 and count2 under lock2.

There are several interesting things to notice about the locking behavior in this program. First, observe that though the variable local is accessed both in the parent thread (lines 13,18) and its child thread thread1 (via the alias *y on line 26), no race is possible despite the lack of synchronization. This is because these accesses cannot occur simultaneously, because the parent only accesses local before the thread for thread1 is created, and never afterward. Thus both accesses are local to a particular thread. Second, tracking of lock acquires and releases must be flow-sensitive, so we know that the access on line 33 is guarded by a lock, and the access on line 35 is not. Lastly, the atomic_inc function is called twice (lines 42–43) with two different locks and integer pointers. We need context sensitivity to avoid conflating these two calls, which would lead to false alarms.



(a) Label flow graph



(b) Abstract control flow graph

Fig. 3. Constraint graphs generated for example in Fig. 2

2.1 Labeling and Constraint Generation

The first phase of LOCKSMITH is *labeling and constraint generation*, which traverses the CIL CFG and generates two key abstractions that form the basis of subsequent analyses: *label flow constraints*, to model the flow of data within the program, and *abstract control flow constraints*, to model the sequencing of key actions and relate them to the label flow constraints. Because a set of label flow constraints can be conveniently visualized as a graph, we will often refer to them as a *label flow graph*, and do likewise for a set of abstract control flow constraints.

2.1.1 *Label flow graph.* Fig. 3(a) shows the label flow graph for the example from Fig. 2. Nodes are static representations of the run-time memory locations (addresses) that contain locks or other data. Edges represent the "flow" of data through the program [Mossin 1996; Rehof and Fähndrich 2001; Kodumal and Aiken 2005], e.g., according to assignment statements or function calls. The source of a

path in the label flow graph is an allocation site, e.g., it is the address of a global or local variable (e.g., &lock1, &count1, or &local in Fig. 2), or the representation of a program line at which a malloc() occurs. We distinguish addresses of locks from those of other data (which may be subject to races); generally speaking we refer to the former using metavariable ℓ and the latter using metavariable ρ.

LOCKSMITH's label flow analysis is *field-sensitive* when modeling C struct types, in which each field of each instance of a struct is modeled separately. We found that field sensitivity significantly improves precision. To make our algorithm sufficiently scalable, we modeled fields lazily [Foster et al. 2006]—only if (and when) a field was actually accessed by the program does LOCKSMITH model it, as opposed to eagerly tracking each field of a given instance from the time the instance is created. We found that over all benchmarks only 35%, on average, of the declared fields of struct variables in the program are actually accessed, and so the lazy approach afforded significant savings.

LOCKSMITH also tries to model C's `void*` types precisely, yet efficiently. In our final design, when a `void*` pointer might point to two different types, we assume that it is not used to cast between them, but rather that the programmer always casts the `void*` pointer to the correct type before using it (in the style of an untagged union). This is an unsound assumption that might possibly mask a race. However, we found it to greatly increase the precision of the analysis, and is usually true for most C programs. We also tried two sound but less precise alternative strategies. First, and most conservatively, if a type is cast to `void*`, we conflate *all* pointers in that type with each other and the `void*`. While sound, this technique is quite imprecise, and the significant amount of false aliasing it produces degrades LOCKSMITH's subsequent analyses. A second alternative we considered behaves in exactly the same way, but only when more than one type is cast to the same `void*` pointer. Assuming a given `void*` is only cast to/from a single type, we can relate any pointers occurring within the type to the particular type instances cast to the `void*`, as if the type was never cast to `void*` in the first place. We found that approximately one third of all `void*` pointers in our benchmarks alias one type, so this strategy increased the precision compared to simply conflating all pointers casted to a `void*`. Nevertheless, we found that our final design is more precise *and* more efficient, in that it prunes several superficial or imprecise constraints.

To achieve context sensitivity, we incorporate additional information about function calls into the label flow graph. Call and return edges corresponding to a call indexed by i in the program are labeled with $(i$ and $)i$, respectively. During constraint resolution, we know that two edges correspond to the same call only if they are labeled by the same index. For example, in Fig. 3(a) the edges from &lock1 and &count1 are labeled with $(4$ since they arise from the call on line 42, and analogously the edges from &lock2 and &count2 are labeled with $(5$. We use a variation on *context-free language (CFL) reachability* to compute the flow of data through the program [Pratikakis et al. 2006b]. In this particular example, since count is accessed with lock held, we would discover that count$i$ is accessed with lock$i$ held for $i \in 1..2$. Without the labeled edges, we could not distinguish the two call sites, and LOCKSMITH would lose precision. In particular, LOCKSMITH would think that lock could be either lock1 or lock2, and thus we would not know which one was held

at the access to count, causing us to report a potential data race on line 7.

Section 3 discusses the label flow analysis in detail, not considering context sensitivity, while Section 5 discusses extensions to this analysis to handle struct and `void*` types. We initially present context-insensitive algorithms for each LOCKSMITH phase, and discuss context sensitivity for all parts in Section 6.

2.1.2 *Abstract control flow graph.* Fig. 3(b) shows the *abstract control flow graph* (ACFG) for the example from Fig. 2. Nodes in the ACFG capture operations in the program that are important for data race detection, and relate them to the labels from the label flow graph. ACFGs contain 7 kinds of nodes (the notation $[n]$ next to each node indicates the line number n from which the node is induced). $\mathsf{NewL}(\ell)$ represents a statement that creates a new lock at location ℓ, and $\mathsf{Acq}(\ell)$ and $\mathsf{Rel}(\ell)$ denote the acquire and release, respectively, of the lock ℓ. Reads and writes of memory location ρ are represented by $\mathsf{Acc}(\rho)$. Thread creation is indicated by Fork nodes, which have two successors: the next statement in the parent thread, and the first statement of the child thread's called function. The edge to the latter is annotated with an index just as in the label flow graph, to allow us to relate the two graphs. For example, the child edge for the Fork corresponding to line 19 is labeled with $(1$, which is the same annotation used for the edge from &local to a in the label flow graph. Lastly, function calls and returns are represented by Call and Ret nodes in the graph. For call site i, we label the edge to the callee with $(i$, and we label the return edge with $)i$, again to allow us to relate the label flow graph with the ACFG. The edges from a Call to the corresponding Ret allow us to flow information "around" callees, often increasing precision; we defer discussion of this feature to Section 3.3.

In addition to label flow constraints and the abstract control flow graph, the first phase of LOCKSMITH also generates *linearity constraints* and *contextual effect constraints*, which are discussed below (Section 2.5).

2.2 Sharing Analysis

The next LOCKSMITH phase determines the set of locations that could be potentially simultaneously accessed by two or more threads during a program's execution. We refer to these as the program's *shared locations*. We limit subsequent analysis for possible data races to these shared locations. In particular, if a location is *not* shared, then it need not be consistently accessed with a particular lock held. Moreover, if an access site (that is, a read or a write through a pointer) never targets a shared variable, it need not be considered by the analysis.

As shown in Fig. 1, this phase takes as input *contextual effect constraints*, which are also produced during labeling and constraint generation. In standard effect systems [Talpin and Jouvelot 1994], the effect of a program statement is the set of locations that may be accessed (read or written) when the statement is executed. Our contextual effect system additionally determines, for each program state, the *future effect*, which contains the locations that may be accessed by the remainder of the current thread. To compute the shared locations in a program, at each thread creation point we intersect the standard effect of the created thread with the future effect of the parent thread. If a location is in both effects, then the location is shared. Note that the future effect of a thread includes the effect of any threads it

creates, so the future effect of the parent includes the effects of any threads that the parent creates later.

For example, consider line 19 in Fig. 2. The spawned thread1 has standard effect {&local}. The parent thread by itself has no future effect, since it accesses no interesting variables. However, it spawns two child threads which themselves access count1 and count2. Therefore, the future effect at line 19 is {&count1, &count2}. Since {&local1}∩{&count1, &count2} = ∅, there are no shared locations created by this fork. In particular, even though local was accessed in the past by the parent thread (line 18), our sharing analysis correctly determines all its accesses to be thread local.

On the other hand, consider line 20. Here the effect of the spawned thread2 is {&count1, &count2}, and the future effect at line 20 is also {&count1, &count2}. Thus we determine from this call that count1 and count2 are shared. Notice that here it was critical to include the effect of thread3 when computing the future effect of the parent, since the parent thread itself does not access anything interesting.

In our implementation, we also distinguish read and write effects, and only mark a location ρ as shared if at least one of the accesses to ρ in the intersected effects is a write. In other words, we do not consider locations that are only read by multiple threads to be shared, and LOCKSMITH does not generate race warnings for them. For that reason, we do not need to differentiate between read and write accesses in the $\mathsf{Acc}(\rho)$ nodes of the ACFG.

The analysis just described only determines if a location is *ever* shared. It could be that a location starts off as thread local and only later becomes shared, meaning that its initial accesses need not be protected by a consistent lock, while subsequent ones do. For example, notice that &count1 in Fig. 2 becomes shared due to the thread creations at lines 20 and 21, since both thread2 and thread3 access it. So while the accesses at lines 33 and 27 (via line 42) must consider &count1 as shared, &count1 would not need to be considered shared if it were accessed at, say, line 19. We use a simple dataflow analysis to distinguish these two cases, and thus avoid reporting a false alarm in the latter case. Section 4 presents the sharing analysis and this variant in more detail.

2.3 Lock State Analysis

In the next phase, LOCKSMITH computes the state of each lock at every program point. To do this, we use the ACFG to compute the set of locks ℓ held before and after each statement.

In the ACFG in Fig. 3(b), we begin at the entry node by assuming all locks are released. In the subsequent discussion, we write A_i for the set of locks that are definitely held after statement i. Since statements 15–21 do not affect the set of locks held, we have $A_{15} = A_{16} = A_{18} = A_{19} = A_{20} = A_{21} = A_{\mathsf{Entry}} = \emptyset$.

We continue propagation for the control flow of the three created threads. Note that even if a lock is held at a fork point, it is not held in the new thread, so we should not propagate the set of held locks along the child Fork edge. For thread1, we find simply that $A_{26} = \emptyset$. For thread2, we have $A_{32} = A_{33} = \{\&\mathsf{lock1}\}$, and $A_{34} = A_{35} = \emptyset$. And lastly for thread3, we have $A_{42} = A_{43} = A_8 = \emptyset$ and $A_6 = A_7 = \{\mathsf{lock}\}$. Notice that this last set contains the name of the formal parameter lock. When we perform correlation inference, discussed next, we will

need to translate information about lock back into information about the actual arguments at the two call sites. Note that for the lock state and the subsequent correlation analyses to be sound, we need to reason about the linearity of locks (introduced in Section 2.5). We do not need to reason about the linearity of shared memory locations the same way, because contrary to locks, any imprecision on the aliasing of memory locations only leads to a more conservative analysis.

2.4 Correlation Inference

The next phase of LOCKSMITH is correlation inference, which is the core race detection algorithm. For each shared variable, we intersect the sets of locks held at all its accesses. We call this the *guarded-by* set for that location, and if the set is empty, we report a possible data race.

We begin by generating initial *correlation constraints* at each $\mathsf{Acc}(\rho)$ node such that ρ may be shared according to the sharing analysis. Correlation constraints have the form $\rho \triangleright \{\ell_1, \ldots, \ell_n\}$, meaning location ρ is accessed with locks ℓ_1 through ℓ_n held. We write C_n for the set of correlation constraints inferred for statement n.

The first access in the program, on line 18, yields no correlation constraints ($C_{18} = \emptyset$) because, as we discussed above, the sharing analysis determines the &local is not a shared variable. Similarly, $C_{26} = \emptyset$ because the only location that "flows to" y in the label flow graph is &local, which is not shared. On line 33, on the other hand, we have an access to a shared variable, and so we initialize $C_{33} = \{\&\mathsf{count1} \triangleright \{\&\mathsf{lock1}\}\}$, using the output of the lock state analysis to report which locks are held. Similarly, $C_{35} = \{\&\mathsf{count2} \triangleright \emptyset\}$, since no locks are held at that access. Finally, $C_7 = \{\mathsf{count} \triangleright \{\mathsf{lock}\}\}$. Here we determine count may be shared because at least one shared variable flows to it in the label flow graph.

Notice that this last correlation constraint is in terms of the local variables of atomic_inc. Thus at each call to atomic_inc, we must instantiate this constraint in terms of the caller's variables. We use an iterative fixpoint algorithm to propagate correlations through the control flow graph, instantiating as necessary until we reach the entry node of main. At this point, all correlation constraints are in terms of the names visible in the top-level scope, and so we can perform the set intersections to look for races. Note that, as is standard for label flow analysis, when we label a syntactic occurrence of malloc() or any other memory allocation or lock creation site, we treat that label as a top-level name.

We begin by propagating C_7 backwards, setting $C_6 = C_7$. Continuing the backwards propagation, we encounter two Call edges. For each call site i in the program, there exists a substitution S_i that maps the formal parameters to the actual parameters; this substitution is equivalent to a polymorphic type instantiation [Rehof and Fähndrich 2001; Pratikakis et al. 2006a]. For call site 4 we have $S_4 = [\mathsf{lock} \mapsto \&\mathsf{lock1}, \mathsf{count} \mapsto \&\mathsf{count1}]$. Then when we propagate the constraints from C_7 backwards across the edge annotated ($_4$, we apply S_4 to instantiate the constraints for the caller. In this case we set $C_{42} = S_4(\{\mathsf{count} \triangleright \{\mathsf{lock}\}\}) = \{\&\mathsf{count1} \triangleright \{\&\mathsf{lock1}\}\}$ and thus we have derived the correct correlation constraint inside of thread3. Similarly, when we propagate C_6 backwards across the edge annotated ($_5$, we find $C_{43} = \{\&\mathsf{count2} \triangleright \{\&\mathsf{lock2}\}\}$.

We continue backwards propagation, and eventually push all the correlations we have mentioned so far back to the entry of main:

&count1 ▷ {&lock1}	from line 33
&count2 ▷ ∅	from line 35
&count1 ▷ {&lock1}	from the call on line 42
&count2 ▷ {&lock2}	from the call on line 43

(Note that there are substitutions for the calls indexed by 1–3, but they do not rename &count*i* or &lock*i*, since those are global names.) We now intersect the lock sets from each correlation, and find that &count1 is consistently correlated with (or guarded by) lock1, whereas &count2 is not consistently correlated with a lock. We then report a race on &count2.

One important detail we have omitted is that when we propagate correlation constraints back to callers, we need to interpret them with respect to the "closure" of the label flow graph. For example, given a constraint x ▷ {&lock}, if &y flows to x in the label flow graph, then we also derive &y ▷ {&lock}. More information about the closure computation can be found elsewhere [Pratikakis et al. 2006b].

Propagating correlation constraints backwards through the ACFG also helps to improve the reporting of potential data races. In our implementation, we also associate a *program path*, consisting of a sequence of file and line numbers, with each correlation constraint. When we generate an initial constraint at an access in the ACFG, the associated path contains just the syntactic location of that access. Whenever we propagate a constraint backwards across a Call edge, we prepend the file and line number of the call to the path. In this way, when the correlation constraints reach the main, they describe a path of function calls from main to the access site, essentially capturing a stack trace of a thread at the point of a potentially racing access, and developers can use these paths to help understand error messages.

Section 3.3.2 presents the algorithm for solving correlation constraints and inferring all correlations in the program. Since correlation analysis is an iterative fixpoint computation, in which we iteratively convert local names to their global equivalents, we compute correlations using the same framework we used to infer the state of locks.

2.5 Linearity and Escape Checking

The constraint generation phase also creates *linearity constraints*, which we use to ensure that a static lock name ℓ used in the analysis never represents two or more run-time locks that are simultaneously live. Without this assurance, we could not model lock acquire and release precisely. That is, suppose during the lock state analysis we encounter a node Acq(ℓ). If ℓ is non-linear, it may represent more than one lock, and thus we do not know which one will actually be acquired by this statement. On the other hand, if ℓ is linear, then it represents exactly one location at run time, and hence after Acq(ℓ) we may assume that ℓ is acquired.

Lock ℓ could be non-linear for a number of reasons. Consider, for example, a linked list data structure where each node of the linked list contains a lock, meant to guard access to the data at that node. Standard label flow analysis will label each element in such a recursive structure with the same name ρ whose pointed-to memory is a record containing some lock ℓ. Thus, ρ statically represents arbitrarily many run-time locations, and consequently ℓ represents arbitrarily many locks.

	Benchmark	Size (LOC)	Time (sec)	Warnings	Not-guarded	Races	Race/Total locations	%
POSIX thread programs	aget	1,914	0.85	62	31	31	62/352	(17.61)
	ctrace	2,212	0.59	10	9	2	10/311	(3.22)
	engine	2,608	0.88	7	0	0	7/410	(1.71)
	knot	1,985	0.78	12	8	8	12/321	(3.74)
	pfscan	1,948	0.46	6	0	0	6/240	(2.50)
	smtprc	8,624	5.37	46	1	1	46/1079	(4.26)
Linux device drivers	3c501	17,443	9.18	15	5	4	15/408	(3.68)
	eql	16,568	21.38	35	0	0	35/270	(12.96)
	hp100	20,370	143.23	14	9	8	14/497	(2.82)
	plip	19,141	19.14	42	11	11	44/466	(9.44)
	sis900	20,428	71.03	6	0	0	6/779	(0.77)
	slip	22,693	16.99	3	0	0	3/382	(0.99)
	sundance	19,951	106.79	5	1	1	5/753	(0.66)
	synclink	24,691	1521.07	139	2	0	139/1298	(10.71)
	wavelan	20,099	19.70	10	1	1	10/695	(1.44)
	Total	200,675	1937.44	412	78	67	414/8261	(5.01)

Fig. 4. Benchmarks

With such a representation, a naive analysis would not complain about a program that acquires a lock contained in one list element but then accesses data present in other elements.

To be conservative, LOCKSMITH treats locks ℓ such as these as *non-linear*, with the consequence that nodes $\mathsf{Acq}(\ell)$ and $\mathsf{Rel}(\ell)$ of such non-linear ℓ are ignored. This approach solves the problem of missing potential races, but is more likely to generate false positives, e.g., when there is an access that is actually guarded by ℓ at run time. LOCKSMITH addresses this issue by using user-specified *existential types* to allow locks in data structures to sometimes be linear, and includes an escape checking phase to ensure existential types are used correctly. We refer the reader to related papers for further discussion on linearity constraints [Pratikakis et al. 2006b] and how they can be augmented with existential quantification [Pratikakis et al. 2006a]. Note that the two locks used in the example of Fig. 2 are linear, since they can only refer to one run-time lock.

2.6 Soundness Assumptions

C's lack of strong typing makes reasoning about C programs difficult. For example, because C programs are permitted to construct pointers from arbitrary integers, it can be difficult to prove that dereferencing such a pointer will not race with memory accessed by another thread. Nevertheless, we believe LOCKSMITH to be sound as long as the C program under analysis has certain characteristics; assuming these characteristics is typical of C static analyses. In particular, we assume that

—the program does not contain memory errors such as double frees or dereferencing of dangling pointers.

—different calls to `malloc()` return different memory locations.

—the program manipulates locks and threads only using the provided API.

—variables that may hold values of more than one type—in particular, `void*` pointers, untagged `union`s, and `va_list` lists of optional function arguments—are used

in a type-safe manner; i.e., each read of such a multi-typed value presumes the type that was last written.[1]

—there is no pointer arithmetic except for indexing arrays, and all array indexing is within bounds.

—`asm` blocks only access what is reachable from their parameters, respecting their types.

—integers are not cast to pointers.

—no non-local control flow such as signal handlers or `setjmp`/`longjmp`.

LOCKSMITH prints a warning for code that may introduce unsoundness.

2.7 Results

Fig. 4 summarizes the results of running LOCKSMITH on a set of benchmarks varying in size, complexity and coding style. The results shown correspond to the default configuration for all LOCKSMITH analyses, in particular: context-sensitive, field-sensitive label flow analysis, with lazy field propagation and no conflation under `void*`; flow- and context-sensitive sharing analysis; and context-sensitive lock state analysis and correlation inference.

The first part of the table presents a set of applications written in C using POSIX threads, whereas the second part of the table presents the results for a set of network drivers taken from the Linux kernel, written in GNU-extended C using kernel threads and spinlocks. The first column gives the benchmark name and the second column presents the number of preprocessed and merged lines of code for every benchmark. We used the CIL merger to combine all the code for every benchmark into a single C file, also removing comments and redundant declarations. The next column lists the running time for LOCKSMITH. Experiments in this paper were performed on a dual-core, 3GHz Pentium D CPU with 4GB of physical memory. All times reported are the median of three runs. The fourth column lists the total number of warnings (shared locations that are not protected by any lock) that LOCKSMITH reports. The next column lists how many of those warnings correspond to cases in which shared memory locations are not protected by any lock, as determined by manual inspection. The sixth column lists how many of those we believe correspond to races. Note that in some cases there is a difference between the unguarded and races columns, where an unguarded location is not a race. These are caused by the use of other synchronization constructs, such as pthread_join, semaphores, or inline atomic assembly instructions, which LOCKSMITH does not model. Finally, the last column presents the number of allocated memory locations that LOCKSMITH reported as unprotected, versus the total number of allocated locations, where we consider struct fields to be distinct memory locations. Note that the number of unprotected memory locations is different from the number of race warnings reported in the fourth column. This is because a LOCKSMITH race warning might involve several concrete memory locations, when they are aliased.

[1] LOCKSMITH can be configured to treat `void*` pointers and `va_list` argument lists conservatively, without assuming that they are type-safe. However, this introduces a lot of imprecision, so it is not the default behavior.

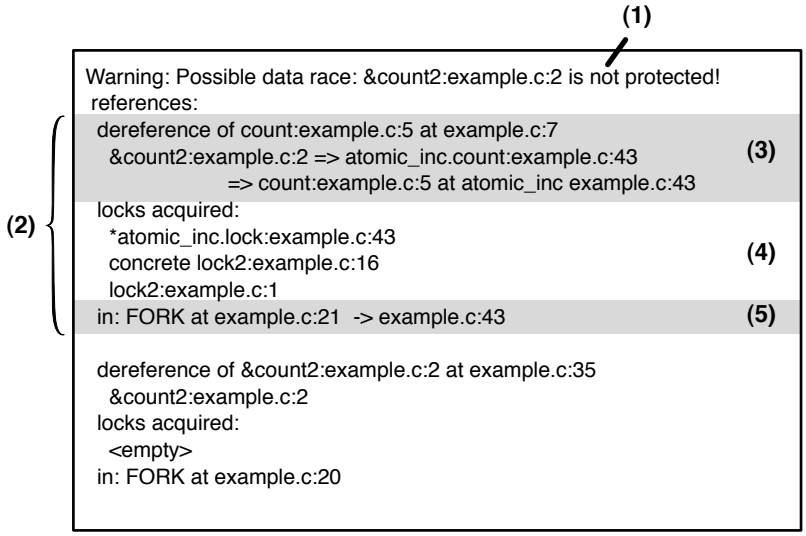


Fig. 5. Sample LOCKSMITH warning. Highlighting and markers added for expository purposes.

2.7.1 *Warnings.* Each warning produced by LOCKSMITH contains information to explain the potential data race to the user. Fig. 5 shows a sample warning from the analysis of the example program shown in Fig. 2, stored in file example.c. The actual output of LOCKSMITH is pure text; here we have added some highlighting and markers (referred to below) for expository purposes.

LOCKSMITH issues one warning per allocation site that is shared but inconsistently (or un-)protected. In this example, the suspect allocation site is the contents of the global variable count2, declared on line 2 of file example.c (1). After reporting the allocation site, LOCKSMITH then describes each syntactic access of the shared location.

The text block indicated by (2) describes the first access site at line 7, accessing variable count which is declared at line 5. The other text block shown in the error report (each block is separated by a newline) corresponds to a different access site. Within a block, LOCKSMITH first describes why the expression that was actually accessed aliases the shared location (3). In this case, the shared location &count2 "flows to" (indicated by =>) the argument count passed to the function call of atomic_inc at line 43 (written as atomic_inc.count), in the label flow graph. That, in turn, flows to the formal argument count of the function, declared at line 5, due to the invocation at line 43. If there is more than one such path in the label flow graph, we list the shortest one.

Next, LOCKSMITH prints the set of locks held at the access (4). We specially designate *concrete* lock labels, which correspond to variables initialized by function pthread_mutex_init(), from aliases of those variables. Aliases are included in the error report to potentially help the programmer locate the relevant pthread_mutex_lock() and pthread_mutex_unlock() statements. In this case, the second lock listed is a concrete lock created at line 16 and named lock2, after the variable that stores the result of pthread_mutex_init(). The global variable lock2 itself, listed third, is

a different lock that aliases the concrete lock created at line 16. There is also an additional alias listed first, the argument ∗lock of the call to function atomic_inc at line 43 (denoted as ∗atomic_inc.lock : example.c : 43). We do not list aliasing paths for the lock sets, because we list all the aliases, and also printing the paths between them would only add confusion by replicating the lock aliases' names many times.

Finally, LOCKSMITH gives stack traces leading up to the access site (5). Each trace starts with the thread creation point (either a call to pthread_create() or the start of main()), followed by a list of function invocations. For this access site, the thread that might perform the access is created at line 21 and then makes a call at example.c : 43 to the function that contains the access. We generate this information during correlation inference, as we propagate information from the access point backwards through the ACFG (Section 3.3.2).

The other dereference listed has the same structure. Notice that the intersection of the held lock sets of the two sites is empty, triggering the warning. In this case, the warning corresponds to a true race, caused by the unguarded write to count2 at line 35, listed as the second dereference in the warning message. To check whether a warning corresponds to an actual race, the programmer has to verify that the listed accesses might actually happen simultaneously, including the case where a single access occurs simultaneously in several threads. Also, the programmer would have to verify that the aliasing listed indeed occurs during the program execution, and is not simply an artifact of imprecision in the points-to analysis. Moreover, the programmer must check whether the set of held locks contains additional locks that LOCKSMITH cannot verify are definitely held. Last, but not least, the programmer needs to validate that the location listed in the warning is in fact shared among different threads.

2.7.2 *Races.* We found races in many of the benchmarks. In knot, all of the races are on counter variables always accessed without locks held. These variables are used to generate usage statistics, which races could render inaccurate, but this would not affect the behavior of knot. In aget, most of the races are due to an unprotected global array of shared information. The programmer intended for each element of the array to be thread-local, but a race on an unrelated memory location in the signal handling code can trigger erroneous computation of array indexes, causing races that may trigger a program crash. The remaining two races are due to unprotected writes to global variables, which might cause a progress bar to be printed incorrectly. In ctrace, two global flag variables can be read and set at the same time, causing them to have erroneous values. In smtprc, a variable containing the number of threads is set in two different threads without synchronization. This can result in not counting some threads, which in turn may cause the main thread to not wait for all child threads at the end of the program. The result is a memory leak, but in this case it does not cause erroneous behavior since it occurs at the end of the program. In most of the Linux drivers, the races correspond to integer counters or flags, but do not correspond to bugs that could crash the program, as there is usually a subsequent check that restores the correct value to a variable. The rest of the warnings for the Linux drivers can potentially cause data corruption, although we could not verify that any can cause the kernel to crash.

$$
\begin{array}{rcl}
e & ::= & x \mid v \mid e\ e \mid \mathsf{if0}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
& \mid & (e, e) \mid e.j \mid \mathsf{ref}\ e \mid\ !\, e \mid e := e \\
& \mid & \mathsf{newlock} \mid \mathsf{acquire}\ e \mid \mathsf{release}\ e \mid \mathsf{fork}\ e \\
v & ::= & n \mid \lambda x{:}t.e \mid (v_1, v_2) \\
t & ::= & \mathit{int} \mid t \times t \mid t \to t \mid \mathit{ref}\ (t) \mid \mathit{lock}
\end{array}
$$

Fig. 6. Source language

2.7.3 *False positives.* The majority of false positives are caused by over-approximation in the sharing analysis. The primary reason for this is conservatism in the label flow (points-to) analysis, which can cause many thread-local locations to be spuriously conflated with shared locations. Since thread-local memory need not be, and usually is not, protected by locks, this causes many false warnings of races on those locations. Overly conservative aliasing has several main causes: structural subtyping where the same data is cast between two different types (e.g. different `struct`s that share a common prefix), `asm` blocks, casting to or from numerical types, and arrays, in decreasing order of significance. One approach to better handling structural subtyping may be adapting physical subtyping [Siff et al. 1999] to LOCKSMITH. Currently, when a `struct` type is cast to a different `struct` type, LOCKSMITH does not compute field offsets to match individual fields, but rather conservatively assumes that all labels of one type could alias all labels of the other.

The second largest source of false positives in the benchmarks is the flow sensitivity of the sharedness property, in more detail than our current flow-sensitive sharing propagation can capture. Specifically, any time that a memory location might be accessed by two threads, we consider it shared immediately when the second thread is created. However, in many cases thread-local memory is first initialized locally, and then becomes shared indirectly, e.g., via an assignment to a global or otherwise shared variable. We eliminate some false positives using a simple intraprocedural uniqueness analysis—a location via a unique pointer as determined by this analysis is surely not shared—but it is too weak for many other situations.

Roadmap. In the remainder of this paper, we discuss the components just described in more detail, starting from its core analysis engine (Section 3), with separate consideration of lock state analysis (Section 3.3.1), correlation inference (Section 3.3.2), the sharing analysis (Section 4), techniques for effectively modeling C `struct` and `void*` types (Section 5), and extensions to enable context-sensitive analysis (Section 6). A detailed discussion of related work on data race detection is presented in Section 7. For many of LOCKSMITH's analysis components, we implemented several possible algorithms, and measured the algorithms' effects on the precision and efficiency of LOCKSMITH. By combining a careful exposition of LOCKSMITH's inner workings with such detailed measurements, we have endeavored to provide useful data to inform further developments in the static analysis of C programs (multithreaded or otherwise). LOCKSMITH is freely available on the web (`http://www.cs.umd.edu/projects/PL/locksmith`).

3. LABELING AND CONSTRAINT GENERATION

We present LOCKSMITH's key algorithms on the language in Fig. 6. This language extends the standard lambda calculus, which consists of variables x, functions $\lambda x{:}t.e$ (where the argument x has type t), and function application $e\ e$. To model conditional control flow, we add integers n and the conditional form $\mathsf{if0}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, which evaluates to e_1 if e_0 evaluates to 0, and to e_2 otherwise. To model structured data (i.e., C structs) we introduce pairs (e, e) along with projection $e.j$. The latter form returns the jth element of the pair ($j \in 1, 2$). We model pointers and dynamic allocation using references. The expression $\mathsf{ref}\ e$ allocates a fresh memory location m, initializing it with the result of evaluating e and returning m. The expression $!\,e$ reads the value stored in memory location e, and the expression $e_1 := e_2$ updates the value in location e_1 with the result of evaluating e_2.

We model locks with three expressions: newlock dynamically allocates and returns a new lock, and acquire e and release e acquire and release, respectively, lock e. Our language also includes the expression fork e, which creates a new thread that evaluates e in parallel with the current thread. The expression fork e is asynchronous, i.e., it returns to the parent immediately without waiting for the child thread to complete.

Source language types t include the integer type *int*, pair types $t \times t$, function types $t \to t$, reference (or pointer) types *ref* (t), and the type *lock* of locks. Note that our source language is monomorphically typed and that, in a function $\lambda x{:}t.e$, the type t of the formal argument x is supplied by the programmer. This matches C, which includes programmer annotations on formal arguments. If we wish to apply LOCKSMITH to a language without these annotations, we can always apply standard type inference to determine such types as a preliminary step.

3.1 Labeling and Constraint Generation

As discussed in Section 2, the first stage of LOCKSMITH is *labeling and constraint generation*, which produces both label flow constraints, to model memory locations and locks, and abstract control flow constraints, to model control flow. We specify the constraint generation phase using type inference rules. We discuss the label flow constraints only briefly here, and refer the reader to prior work [Mossin 1996; Fähndrich et al. 2000; Rehof and Fähndrich 2001; Kodumal and Aiken 2004; Johnson and Wagner 2004; Pratikakis et al. 2006a], including the LOCKSMITH conference paper [Pratikakis et al. 2006b], for more details. In our implementation, we use BANSHEE [Kodumal and Aiken 2005], a set-constraint solving engine, to represent and solve label flow constraints. For the time being, we present a purely monomorphic (context-insensitive) analysis; Section 6 discusses context sensitivity.

We extend source language types t to *labeled types* τ, defined by the grammar at the top of Fig. 7(a). The type grammar is mostly the same as before, with two main changes. First, reference and lock types now have the forms $\mathit{ref}^{\,\rho}(\tau)$ and lock^{ℓ}, where ρ is an abstract location and ℓ is an abstract lock. As mentioned in the last section, each ρ and ℓ stands for one or more concrete, run-time locations. Second, function types now have the form $(\tau, \phi) \to^{\ell} (\tau', \phi')$, where τ and τ' are the domain and range type, and ℓ is a lock, discussed further below. In this function type, ϕ and ϕ' are *statement labels* that represent the entry and exit node of the function.

$$
\begin{array}{rcl}
\tau & ::= & \mathit{int} \mid \tau \times \tau \mid (\tau, \phi) \rightarrow^{\ell} (\tau, \phi) \mid \mathit{ref}^{\,\rho}(\tau) \mid \mathit{lock}^{\ell} \\
C & ::= & C \cup C \mid \tau \leq \tau \mid \rho \leq \rho \mid \ell \leq \ell \mid \phi \leq \phi \mid \phi : \kappa \\
\kappa & ::= & \mathsf{Acc}(\rho) \mid \mathsf{NewL}(\ell) \mid \mathsf{Acq}(\ell) \mid \mathsf{Rel}(\ell) \\
 & \mid & \mathsf{Fork} \mid \mathsf{Call}(\ell, \phi) \mid \mathsf{Ret}(\ell, \phi)
\end{array}
$$

$$
\begin{array}{rcl}
\rho & \in & \textit{abstract locations} \\
\ell & \in & \textit{abstract locks} \\
\phi & \in & \textit{abstract statement labels}
\end{array}
$$

$$
\begin{array}{rclrcll}
\langle\langle \mathit{int} \rangle\rangle & = & \mathit{int} & \langle\langle t_1 \rightarrow t_1 \rangle\rangle & = & (\langle\langle t_1 \rangle\rangle, \phi_1) \rightarrow^{\ell} (\langle\langle t_2 \rangle\rangle, \phi_2) & \phi_1, \phi_2, \ell \textit{ fresh} \\
\langle\langle t_1 \times t_2 \rangle\rangle & = & \langle\langle t_1 \rangle\rangle \times \langle\langle t_2 \rangle\rangle & \langle\langle \mathit{ref}\ (t) \rangle\rangle & = & \mathit{ref}^{\,\rho}(\langle\langle t \rangle\rangle) & \rho \textit{ fresh} \\
 & & & \langle\langle \mathit{lock} \rangle\rangle & = & \mathit{lock}^{\ell} & \ell \textit{ fresh}
\end{array}
$$

$\langle\langle \tau \rangle\rangle = \tau'$ where τ' is τ with fresh ρ, ℓ, ϕ's, as above

$$C \vdash \phi \leq (\phi' : \kappa) \equiv C \vdash \phi \leq \phi' \text{ and } C \vdash \phi' : \kappa \text{ and } \phi' \textit{ fresh}$$

(a) Auxiliary definitions

$$\textsc{Var}\ \frac{}{C; \phi; \Gamma, x : \tau \vdash x : \tau; \phi} \qquad \textsc{Int}\ \frac{}{C; \phi; \Gamma \vdash n : \mathit{int}; \phi}$$

$$\textsc{Pair}\ \frac{C; \phi; \Gamma \vdash e_1 : \tau_1; \phi_1 \quad C; \phi_1; \Gamma \vdash e_2 : \tau_2; \phi_2}{C; \phi; \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2; \phi_2} \qquad \textsc{Proj}\ \frac{C; \phi; \Gamma \vdash e : \tau_1 \times \tau_2; \phi' \quad j \in 1, 2}{C; \phi; \Gamma \vdash e.j : \tau_j; \phi'}$$

$$\textsc{Ref}\ \frac{C; \phi; \Gamma \vdash e : \tau; \phi' \quad \rho \textit{ fresh}}{C; \phi; \Gamma \vdash \mathsf{ref}\ e : \mathit{ref}^{\,\rho}(\tau); \phi'} \qquad \textsc{Deref}\ \frac{C; \phi; \Gamma \vdash e_1 : \mathit{ref}^{\,\rho}(\tau); \phi_1 \quad C \vdash \phi_1 \leq (\phi' : \mathsf{Acc}(\rho))}{C; \phi; \Gamma \vdash\ !\, e_1 : \tau; \phi'}$$

$$\textsc{Assign}\ \frac{C; \phi; \Gamma \vdash e_1 : \mathit{ref}^{\,\rho}(\tau_1); \phi_1 \quad C; \phi_1; \Gamma \vdash e_2 : \tau_2; \phi_2 \quad C \vdash \tau_2 \leq \tau_1 \quad C \vdash \phi_2 \leq (\phi' : \mathsf{Acc}(\rho))}{C; \phi; \Gamma \vdash e_1 := e_2 : \tau_2; \phi'}$$

$$\textsc{Newlock}\ \frac{C \vdash \phi \leq (\phi' : \mathsf{NewL}(\ell)) \quad \ell \textit{ fresh}}{C; \phi; \Gamma \vdash \mathsf{newlock} : \mathit{lock}^{\ell}; \phi'} \qquad \textsc{Acquire}\ \frac{C; \phi; \Gamma \vdash e : \mathit{lock}^{\ell}; \phi' \quad C \vdash \phi' \leq (\phi'' : \mathsf{Acq}(\ell))}{C; \phi; \Gamma \vdash \mathsf{acquire}\ e : \mathit{int}; \phi''}$$

$$\textsc{Release}\ \frac{C; \phi; \Gamma \vdash e : \mathit{lock}^{\ell}; \phi' \quad C \vdash \phi' \leq (\phi'' : \mathsf{Rel}(\ell))}{C; \phi; \Gamma \vdash \mathsf{release}\ e : \mathit{int}; \phi''}$$

$$\textsc{Fork}\ \frac{C; \phi'; \Gamma \vdash e : \tau; \phi'' \quad C \vdash \phi \leq (\phi' : \mathsf{Fork})}{C; \phi; \Gamma \vdash \mathsf{fork}\ e : \mathit{int}; \phi} \qquad \textsc{Lam}\ \frac{\tau = \langle\langle t \rangle\rangle \quad C; \phi_\lambda; \Gamma, x : \tau \vdash e : \tau'; \phi'_\lambda \quad \phi_\lambda \textit{ fresh} \quad \ell = \{\text{locks } e \text{ may access}\}}{C; \phi; \Gamma \vdash \lambda x{:}t.e : (\tau, \phi_\lambda) \rightarrow^{\ell} (\tau', \phi'_\lambda); \phi}$$

$$\textsc{App}\ \frac{\begin{array}{c} C; \phi; \Gamma \vdash e_1 : (\tau, \phi_\lambda) \rightarrow^{\ell} (\tau', \phi'_\lambda); \phi_1 \quad \phi_1; \Gamma \vdash e_2 : \tau_2; \phi_2 \quad C \vdash \tau_2 \leq \tau \\ C \vdash \phi_2 \leq (\phi_3 : \mathsf{Call}(\ell, \phi')) \quad C \vdash \phi_3 \leq \phi_\lambda \quad C \vdash \phi'_\lambda \leq (\phi' : \mathsf{Ret}(\ell, \phi_3)) \end{array}}{C; \phi; \Gamma \vdash e_1\ e_2 : \tau'; \phi'}$$

$$\textsc{Cond}\ \frac{\begin{array}{c} C; \phi; \Gamma \vdash e_0 : \mathit{int}; \phi_0 \quad C; \phi_0; \Gamma \vdash e_1 : \tau_1; \phi_1 \quad C; \phi_0; \Gamma \vdash e_2 : \tau_2; \phi_2 \\ \tau = \langle\langle \tau_1 \rangle\rangle \quad C \vdash \tau_1 \leq \tau \quad C \vdash \tau_2 \leq \tau \\ C \vdash \phi_1 \leq \phi' \quad C \vdash \phi_2 \leq \phi' \quad \phi' \textit{ fresh} \end{array}}{C; \phi; \Gamma \vdash \mathsf{if0}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau; \phi'}$$

(b) Type inference rules

Fig. 7. Labeling and Constraint Generation Rules

We will usually say *statement* ϕ instead of *statement label* ϕ when the distinction is made clear by the use of a ϕ metavariable.

During type inference, our type rules generate *constraints* C, including *flow constraints* of the form $\tau \leq \tau'$, indicating a value of type τ *flows to* a position of type τ'. Flow constraints among types are ultimately reduced to flow constraints $\rho \leq \rho'$ and $\ell \leq \ell'$ among locations and locks, respectively. When we draw a set of label flow constraints as a graph, as in Fig. 3(a), ρ's and ℓ's form the nodes, and each constraint $x \leq y$ is drawn as a directed edge from x to y.

We also generate two kinds of constraints that, put together, define the ACFG. Whenever statement ϕ occurs immediately before statement ϕ', our type system generates a constraint $\phi \leq \phi'$. As above, we drawn such a constraint as an edge from ϕ to ϕ'. We generate *kind constraints* of the form $\phi : \kappa$ to indicate that statement ϕ has *kind* κ, where the kind indicates the statement's relevant behavior, as described in Section 2.1. Note that our type rules assign at most one kind to each statement label, and thus we showed only the kinds of statement labels in Fig. 3(b). Statement labels with no kind, including join points and function entries and exits, have no interesting effect.

The bottom half of Fig. 7(a) defines some useful shorthands. The notation $\langle\langle \cdot \rangle\rangle$ denotes a function that takes either a standard type or a labeled type and returns a new labeled type with the same shape but with fresh abstract locations, locks, and statement labels at each relevant position. By *fresh* we mean a metavariable that has not been introduced elsewhere in the typing derivation. We also use the abbreviation $C \vdash \phi \leq (\phi' : \kappa)$, which stands for $C \vdash \phi \leq \phi'$, $C \vdash \phi' : \kappa$, and ϕ' *fresh*. These three operations often go together in our type inference rules.

Fig. 7(b) gives type inference rules that prove judgments of the form $C; \phi; \Gamma \vdash e : \tau; \phi'$, meaning under constraints C and type environment Γ (a mapping from variable names to labeled types), if the preceding statement label is ϕ (the *input statement label*), then expression e has type τ and has the behavior described by statement ϕ' (the *output statement label*). In these rules, the notation $C \vdash C'$ means that C must contain the constraints C'. Viewing these rules as defining a constraint generation algorithm, we interpret this judgment as *generating* the constraint C' and adding it to C.

We discuss the rules briefly. Var and Int are standard and yield no constraints. The output statement labels of these rules are the same as the input statement labels, since accessing a variable or referring to an integer has no effect.

In Pair, we type e_1 with the input statement ϕ for the whole expression, yielding output statement ϕ_1. We then type e_2 starting in ϕ_1 and yielding ϕ_2, the output statement label for the whole expression. Notice we assume a left-to-right order of evaluation. In Proj, we type check the subexpression e, and the output statement label of the whole expression is the output of e.

Ref types memory allocation, associating a fresh abstract location ρ with the newly created updatable reference. Notice that this rule associates ρ with a syntactic occurrence of `ref`, but if that `ref` is in a function, it may be executed multiple times. Hence the single ρ chosen by Ref may stand for multiple run-time locations.

Deref is the first rule to actually introduce a new statement label into the abstract control flow graph. We type e_1, yielding a pointer to location ρ and

an output statement ϕ_1. We then add a new statement ϕ' to the control flow graph, occurring immediately after ϕ_1, and give ϕ' the kind $\mathsf{Acc}(\rho)$ to indicate the dereference. Statement ϕ' is the output of the whole expression. Assign is similar, but also requires the type τ_2 of e_2 be a subtype of the type τ_1 of data referenced by the pointer e_1.

Newlock types a lock allocation, assigning it a fresh abstract lock ℓ, similarly to Ref. Acquire and Release both require that their argument be a lock, and both return some *int* (in C these functions typically return `void`). All three of these rules introduce a new statement label of the appropriate kind into the control flow graph immediately after the output statement label of the last subexpression.

In Fork, the control flow is somewhat different than the other rules, to match the asynchronous nature of fork. At run time, the expression e is evaluated in a new thread. Hence we introduce a new statement ϕ' with the special kind Fork, to mark thread creation, and type e with input statement ϕ'. We sequence ϕ' immediately after ϕ, since that is their order in the control flow. The output statement label of the fork expression as a whole is the same as the input statement ϕ, since no state in the parent changes after the fork.

In Cond, we sequence the subexpressions as expected. We type both e_1 and e_2 with input statement ϕ_0, since either may occur immediately after e_0 is evaluated. We also create a fresh statement ϕ' representing the join point of the condition, and add appropriate constraints to C. Since the join point has no effect on the program state, we do not associate a kind with it. We also join the types τ_1 and τ_2 of the two branches, constraining them to flow to a type τ, which has the same shape as τ_1 but has fresh locations and locks. Note that for the constraints in this rule to be satisfied (as described in the following section), τ_2 must have the same shape as τ_1, and so we could equivalently have written $\tau = \langle\langle \tau_2 \rangle\rangle$.

Lam type checks the function body e in an environment with x bound to τ, which is the standard type t annotated with fresh locations and locks. We create a new statement ϕ_λ to represent the function entry, and use that as the input statement label when typing the function body e. We place ϕ_λ and ϕ'_λ, the output statement label after e has been evaluated, in the type of the function. We also add an abstract lock ℓ to the function type to represent the function's *lock effect*, which is the set of locks that may be acquired or released when the function executes. For each lock ℓ' that either e acquires or releases directly or that appears on the arrow of a function called in e, a separate effect analysis (not shown) generates a constraint $\ell' \leq \ell$. Then during constraint resolution, we compute the set of locks that flow to ℓ to compute the lock effect. We discuss the use of lock effects in Section 2.4. The output statement label for the expression as a whole is the same as the input statement label, since defining a function has no effect.

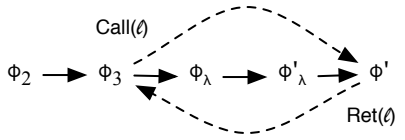
Finally, App requires that e_2's type be a subtype of e_1's domain type. We also add the appropriate statement labels to the control flow graph. Statement ϕ_2 is the output of e_2, and ϕ_λ is the entry node for the function. Thus clearly we need to add control flow from ϕ_2 to ϕ_λ. Moreover, ϕ'_λ is the output statement label of the function body, and that should be the last statement label in the function application as a whole. However, rather than directly inserting ϕ_λ and ϕ'_λ in the control flow graph, we introduce two intermediate statement labels, ϕ_3 just before

$$
\begin{array}{rcl}
C \cup \{int \leq int\} & \Rightarrow & C \\
C \cup \{ref^{\rho}(\tau) \leq ref^{\rho'}(\tau')\} & \Rightarrow & C \cup \{\rho \leq \rho', \tau \leq \tau', \tau' \leq \tau\} \\
C \cup \{lock^{\ell} \leq lock^{\ell'}\} & \Rightarrow & C \cup \{\ell \leq \ell'\} \\
C \cup \{\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'\} & \Rightarrow & C \cup \{\tau_1 \leq \tau_1', \tau_2 \leq \tau_2'\} \\
C \cup \{(\tau_1, \phi_1) \rightarrow^{\ell_1} (\tau_1', \phi_1') \leq (\tau_2, \phi_2) \rightarrow^{\ell_2} (\tau_2', \phi_2')\} & \Rightarrow & C \cup \{\tau_2 \leq \tau_1, \tau_1' \leq \tau_2', \ell_1 \leq \ell_2\} \\
 & \cup & \{\phi_2 \leq \phi_1, \phi_1' \leq \phi_2'\}
\end{array}
$$

$$
\begin{array}{rcl}
C \cup \{\rho \leq \rho', \rho' \leq \rho''\} & \cup \Rightarrow & \{\rho \leq \rho''\} \\
C \cup \{\ell \leq \ell', \ell' \leq \ell''\} & \cup \Rightarrow & \{\ell \leq \ell''\}
\end{array}
$$

Fig. 8. Label flow constraint rewriting rules

the call, and ϕ' just after. Statement ϕ_3 has kind $\mathsf{Call}(\ell, \phi')$, and statement ϕ' has kind $\mathsf{Ret}(\ell, \phi_3)$. Pictorially, the control flow graph looks like the following, where the ϕ's in the kinds of the Call and Ret nodes are drawn with dashed lines:



Using this structure, we can propagate certain dataflow facts "around" functions—i.e., directly from Call to Ret, rather than through the function body—thereby improving precision and gaining some speed up. In particular, we use this for our lock state computation (Section 2.3).

3.2 Label Flow Constraint Resolution

After generating constraints using the rules in Fig. 7, we can then solve the constraints to compute various facts about the analyzed program. We use *flow constraints* to answer questions about which locations and locks are used by various statements in the program, i.e., to perform a label flow analysis. These constraints have the form $c \leq c'$ where c and c' are either locations ρ, locks ℓ, or types τ.[2]

We apply the rewriting rules in Fig. 8 to translate the constraints to a simpler, solved form. The first group of rewriting rules operate on constraints of the form $\tau \leq \tau'$. These rules are standard structural subtyping rules, matching the shapes of the left- and right-hand sides of the constraints and then propagating subtyping to the components in the usual way (e.g., invariant for references and contravariant for function domains) [Pierce 2002]. We will assume that the source program is type correct with respect to the standard types, so that these rewriting rules will never encounter a constraint they cannot reduce further, i.e., in the constraint $\tau \leq \tau'$, the types τ and τ' will always be the same modulo abstract locations and locks.

After applying these rewriting rules, we are left with constraints $\rho \leq \rho'$ and $\ell \leq \ell'$. The remaining rewrite rules add any transitively implied constraints. Here the notation $C \cup \Rightarrow C'$ means we add the constraints C' to C. We define $Sol(C)$ to be the set of constraints computed by exhaustively applying the rules in Fig. 8

[2]We discuss the control flow constraints on ϕ labels in Section 3.3.

to C. We can then define

$$\begin{aligned} \mathit{Flow}(C, \rho) &= \{\rho' \mid \rho' \leq \rho \in \mathit{Sol}(C)\} \\ \mathit{Flow}(C, \ell) &= \{\ell' \mid \ell' \leq \ell \in \mathit{Sol}(C)\} \end{aligned}$$

In other words, $\mathit{Flow}(C, \rho)$ is the set of abstract locations that *flow to* ρ in the constraints C, and similarly for $\mathit{Flow}(C, \ell)$. We can use this information to answer questions about locations and locks in the program. For example, given a dereference site $!\,e$, if type inference assigns e the type $\mathit{ref}^{\,\rho}(\mathit{int})$ and $\mathsf{ref}\ e'$ the type $\mathit{ref}^{\,\rho'}(\mathit{int})$, then if it is possible for e to evaluate to $\mathsf{ref}\ e'$, then $\rho' \in \mathit{Flow}(C, \rho)$. In other words, the set $\mathit{Flow}(C, \rho)$ conservatively models the set of locations that may flow to the reference annotated with ρ.

Consider the example of Section 2. In function thread1() (lines 23–28) the argument a corresponds to a variable with type $\mathit{ref}^{\,\rho_{\&a}}(\mathit{ref}^{\,\rho_a}(\mathit{int}))$ in this language, ignoring for now the special `void*` type. (We follow the convention that the location name is subscripted by the program variable that names the location.) Similarly, the local variable y in thread1() corresponds to a variable with type $\mathit{ref}^{\,\rho_{\&y}}(\mathit{ref}^{\,\rho_y}(\mathit{int}))$. (Because in C variables can be l-values, we consider all variables to be references to the corresponding type, adding an extra level of reference that is implicit in the C program.) In C, variable names are implicitly dereferenced when they occur in a read context. For example, the assignment to y in thread1() (line 24) can be written as $\mathsf{y} = \mathsf{a}$, where the occurrence of y at the left hand side of the assignment denotes the location y whereas the occurrence of a at the right hand side denotes its contents. In our formal language, that assignment corresponds to $y := !\,a$. Typing this assignment with Assign and Deref creates the flow constraint $\mathit{ref}^{\,\rho_a}(\mathit{int}) \leq \mathit{ref}^{\,\rho_y}(\mathit{int})$. Then, the second and first rewriting rules in Fig. 8 solve the constraint reducing it to $\rho_a \leq \rho_y$, and thus $\rho_a \in \mathit{Flow}(C, \rho_y)$.

Note that the constraints in this section are monomorphic and do not include the $(i$ and $)i$ edges we introduced in Section 2 for context sensitivity. We will discuss how to incorporate context sensitivity into this system in Section 6.

3.3 Data Flow Analysis with the Abstract Control Flow Graph

Locksmith uses a generic, mostly standard dataflow analysis engine to compute per-program point information, such as which locks are held, by propagating dataflow facts through the ACFG. To construct a dataflow analysis, the programmer specifies the following characteristics of the target analysis [Aho and Ullman 1977]:

—The direction of the analysis (forwards or backwards)

—The type of the dataflow facts to propagate

—Initial dataflow facts at the program entry, and at each statement label

—A merge function to join dataflow facts

—Transfer functions for each kind of statement label

In the remainder of this section, we discuss two dataflow analyses used by Locksmith—lock state analysis and correlation inference—and then compare the performance of various strategies for implementing the fixpoint computation.

ϕ kind	$Acq_{out}(\phi)$
$\mathsf{Acc}(\rho)$	$Acq_{in}(\phi)$
Fork	$\emptyset$
$\mathsf{NewL}(\ell)$	$Acq_{in}(\phi)$
$\mathsf{Acq}(\ell)$	$Acq_{in}(\phi) \cup Flow(C, \ell)$

ϕ kind	$Acq_{out}(\phi)$
$\mathsf{Rel}(\ell)$	$Acq_{in}(\phi) \setminus Flow(C, \ell)$
$\mathsf{Call}(\ell, \phi')$	$Acq_{in}(\phi) \cap Flow(C, \ell)$
	$Split(\phi') = Acq_{in}(\phi) \setminus Flow(C, \ell)$
$\mathsf{Ret}(\ell, \phi')$	$Acq_{in}(\phi) \cup Split(\phi)$

Fig. 9. Transfer functions for lock state inference

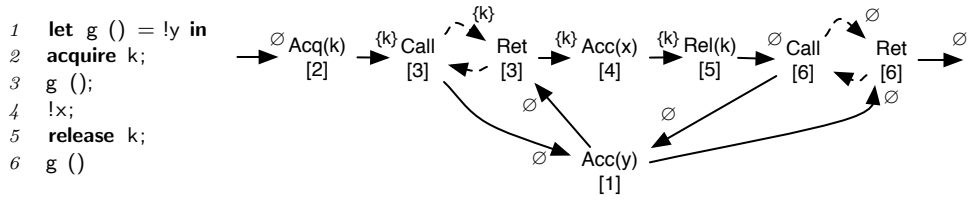


Fig. 10. Splitting the lock state at a function call

3.3.1 *Lock State: Forwards dataflow.* LOCKSMITH's lock state analysis is a forwards dataflow analysis, where the sets of dataflow facts are the sets of locks held. The set of held locks is initially empty for all statement labels, and the merge function is set intersection. Fig. 9 lists the transfer functions for each kind of statement label. $\mathsf{Acc}(\rho)$ and $\mathsf{NewL}(\ell)$ do not alter the lock state, since neither acquires or releases a lock. The transfer function for Fork always returns the empty set of locks, as every new thread starts with all locks released. (Recall from Fig. 7 that the first statement label in a thread has kind Fork.) The transfer functions for $\mathsf{Acq}(\ell)$ and $\mathsf{Rel}(\ell)$ add and remove, respectively, $Flow(C, \ell)$ from the lock state. This latter set includes ℓ and all its aliases. The separate linearity check (mentioned in Section 2.5) ensures this set contains only one run-time lock. In our implementation, we also signal a warning at an attempt to acquire or release a lock that is already acquired or released, respectively.

The transfer function for $\mathsf{Call}(\ell, \phi')$ partitions the held locks into two non-overlapping sets. The transfer function sets the output set of held locks to be the input set intersected with $Flow(C, \ell)$, which contains the lock effect of the function, i.e., the aliases of all locks that may be changed by the called function. It is this intersection that will be propagated into the body of the function. The transfer function saves the remaining held locks in the set $Split(\phi')$. Then the transfer function for $\mathsf{Ret}(\ell, \phi')$ adds the saved locks back to the held set. Due to the way the inference rules generate fresh variables, it is always the case that ϕ' is unique, generated fresh for each call site.

For example, consider the program in Fig. 10. This program calls function g twice, once with k held, and once without holding k. It also accesses x with k held, just after the first call to g. The function g itself accesses y. The right side of the figure shows the ACFG for this example, annotated with the lock state $Acq_{out}(\phi)$ at the end of each edge from statement ϕ. Statement numbers are given below the statement kinds. Initially, no locks are held ($\emptyset$ on the edge to $\mathsf{Acq}(\mathsf{k})$ [2]), and after line 2 the lock state is $\{\mathsf{k}\}$ (shown on the edge from $\mathsf{Acq}(\mathsf{k})$ [2]). Then at

ϕ kind	$Corr_{out}(\phi)$
Acc(ρ) (ρ shared)	$Corr_{in}(\phi) \cup \{\rho \rhd Acq_{out}(\phi)\}$
Fork	$Corr_{in}(\phi)$
NewL(ℓ)	$Corr_{in}(\phi)$

ϕ kind	$Corr_{out}(\phi)$
Acq(ℓ)	$Corr_{in}(\phi)$
Rel(ℓ)	$Corr_{in}(\phi)$
Call(ℓ, ϕ')	$(Corr_{in}(\phi) + Split(\phi')) \cup Corr_{in}(\phi')$
Ret(ℓ, ϕ')	$\emptyset$

Fig. 11. Transfer functions for correlation inference

the call to g, we split the lock state into two parts. Since g does not acquire or release any locks, we propagate $\{k\} \cap \emptyset = \emptyset$ from the Call [3] to Acc(y) [1]. During correlation inference, we will recover the fact that y was accessed with k held. We propagate the other part of the lock state, $\{k\} \setminus \emptyset$, from Call [3] to Ret [3]. Next, after Ret [3], the lock state is $\{k\} \cup \emptyset$, i.e., the locks that flowed "around" g plus the locks that flowed "through" g. Thus at Acc(x) [4], we see that k is held. Continuing through the program, at the next call to g, the empty set of locks is held, and that is propagated into g as before. This time after the Ret no locks are held, and the program continues.

Critically, with this analysis we can discover that x is guarded by k. Imagine if we had not split the lock state. Then we would have no dashed lines in the graph in Fig. 10, and we would directly connect the Call and Ret nodes to Acc(y). But then we would have two edges flowing to Acc(y), one with state $\{k\}$, and one with state $\emptyset$. We would then intersect these sets and decide that no locks were held at the entry to g. That summarization is fine for g, but when we propagate this information, we would decide no locks were held after the Ret statement labels in the ACFG, and thus we would think x was accessed with no lock held.

In essence, by splitting the lock state, we make functions parametric in locks they do not change; similar approaches have been used in other type systems [Smith et al. 2000; Foster et al. 2002]. (We do propagate information about changed locks into the function, since otherwise we would not be able to track the state of those locks correctly.) We have found this kind of lightweight polymorphism critical to LOCKSMITH's precision. It is particularly important for commonly called functions such as printf, which would otherwise almost always cause the lock state to be empty upon their return.

3.3.2 *Correlation Inference: Backwards dataflow.* LOCKSMITH also uses the dataflow analysis engine to implement correlation inference. Recall from Section 2.4 that a correlation constraint has the form $\rho \rhd \{\ell_1, \ldots, \ell_n\}$, where the ℓ_i are the locks that are held during an access to ρ. To generate such constraints the analysis uses a backwards propagation, where the per-ϕ state is a set *Corr* of correlations. Initially the set of correlations is empty for all statement labels, and the merge function is set union.

Fig. 11 shows the transfer rules for correlation inference. Note that since this is a backwards analysis, $Corr_{in}(\phi)$ corresponds to the state after statement ϕ, and $Corr_{out}(\phi)$ corresponds to the state before ϕ.

The transfer function for Acc(ρ) adds $\rho \rhd Acq_{out}(\phi)$ to the set of correlations, where ρ is determined to be shared according to the sharing analysis (Section 4), and $Acq_{out}(\phi)$ was computed by the lock state inference. The transfer functions

for Fork, NewL(ℓ), Acq(ℓ), and Rel(ℓ) simply propagate the set of correlations. The last two transfer functions are the most interesting. Recall that during the lock state computation, any held locks that are not changed by a function are not propagated through the function body. However, any lock that is held for the duration of a function call clearly is correlated with all accesses that occur in the body of the function. Because of this, the transfer function for Call(ℓ, ϕ') adds the set $Split(\phi')$ of held locks that are "hidden" from the function body, to the lock set of every correlation for the function. We define $Corr_{in}(\phi) + Split(\phi')$ to be the set of correlations $\{\rho \rhd (\{\ell_1, \ldots, \ell_n\} \cup Split(\phi'))\}$ where $\rho \rhd \{\ell_1, \ldots, \ell_n\} \in Corr_{in}(\phi)$.

We can apply this transfer function to all correlations caused by accesses during the execution of the invoked function. However, if the called function creates a new thread, the correlations originating from that thread are clearly not protected by the held locks in $Split(\phi')$, even though they are propagated backwards through the function body during the analysis. We handle that case by marking all correlations that have been propagated through a Fork point as *closed*, so that their lock set is not augmented through *Split* nodes.

We also include in the output of Call(ℓ, ϕ') all correlations that occur after the function returns, $Corr_{in}(\phi')$. Then, the transfer function for Ret(ℓ, ϕ') always returns the empty set, as all correlations occurring after the function call are already propagated to the point before it. This speeds up correlation inference, because we need not propagate correlation information into called functions. Also, recall that due to the use of the special call and return kinds in the lock state analysis, we "hide" a set of held locks for every function call. Clearly, as the locks in the split set are held during the function call, they protect all dereferences that occur in this given evaluation of the function body. We therefore need to add that "hidden" set of held locks to the set of correlations that occur in the function. Propagating correlations this way facilitates that, as only the correlations that occur in the function body are propagated through the Call(ℓ, ϕ') node.

To see these transfer functions in action, consider again the program in Fig. 10. Initially the correlation sets are empty, but the transfer functions for accesses Acc(x) [4] and Acc(y) [1] introduce two initial correlations in this program: $\mathsf{x} \rhd \{\mathsf{k}\}$ (generated at Acc(x) [4]) and $\mathsf{y} \rhd \emptyset$ (generated at Acc(y) [1]). The correlation constraint $\mathsf{x} \rhd \{\mathsf{k}\}$ is first propagated backward to Ret [3], then to Call [3], then Acq(k) [2], and then to the beginning of the program. Notice that following the rule for Ret in Fig. 11, we do not propagate this constraint backwards into node [11] in the called function.

There are two backward paths for the second correlation, on y. When we propagate it to Call [3], we add the "hidden" lock k to the correlation, yielding $\mathsf{y} \rhd \{\mathsf{k}\}$ at Call [3]. When we propagate it to Call [6], there are no hidden locks, and so we get correlation $\mathsf{y} \rhd \emptyset$. We propagate both of these constraints unchanged to the start of the program. Thus we have that y is correlated with both $\{\mathsf{k}\}$ and $\emptyset$, and therefore y is not consistently guarded by a lock.

In our implementation, we also associate a call stack with each correlation constraint. When we propagate information through a Call node, we add the name of the called function to the call stack. In this way, once correlations reach the entry of the whole program, we can report not only what locations are correlated

Benchmark	Queue	Stack	Double Stack	Postorder Set	No worklist
aget	0.84	0.82	0.83	0.80	0.85 (5)
ctrace	0.61	0.58	0.60	0.57	0.59 (4)
engine	0.88	0.88	0.87	0.89	0.88 (5)
knot	0.77	0.74	0.76	0.74	0.78 (8)
pfscan	0.43	0.43	0.41	0.43	0.46 (5)
smtprc	5.92	6.77	5.63	4.31	5.37 (7)
3c501	9.03	9.46	9.21	9.39	9.18 (6)
eql	timeout	timeout	timeout	timeout	21.38 (4)
hp100	timeout	timeout	timeout	2524.49	143.23 (15)
plip	30.73	30.21	28.53	25.11	19.14 (5)
sis900	85.07	82.89	84.97	79.37	71.03 (6)
slip	timeout	timeout	timeout	timeout	16.99 (5)
sundance	104.22	108.92	108.59	103.73	106.79 (11)
synclink	timeout	timeout	timeout	timeout	1521.07 (11)
wavelan	19.69	20.08	19.66	19.76	19.70 (6)

Fig. 12. Time (in seconds) to perform correlation inference using several fixpoint strategies. The rightmost column also includes the number of visits.

with which locks, but also on what paths the dereferences occurred. For example, for the code in Fig. 10, if y were shared, we would report a data race, indicating that the accesses were due to the call on line 3 and the call on line 6. We have found that this "path" information makes LOCKSMITH error reports much easier to understand.

3.3.3 *Fixpoint Computation Strategies.* We implemented several strategies for finding a fixpoint of the sets computed by our dataflow analyses. First, we experimented with worklist-based schemes. In these approaches, each time the input set (i.e. the output of predecessors or successors, for a forwards or backwards analysis, respectively) computed for some node ϕ changed, we would add ϕ to the worklist for reconsideration. We tried four particular worklist implementations, discussed in detail by Cooper et al. [2004]: Queue, Stack, Double Stack, and Postorder Set. Initially, our implementations avoided adding duplicate nodes to the worklist, but we found that the cost to detect and eliminate duplicates is comparable to the gain from not processing the additional nodes. Thus, none of the implementations reported here attempt to eliminate duplicate nodes.

Second, we implemented the following simple strategy for backwards (forwards) analysis without a worklist:

(1) Starting from the exit (entry) nodes, perform a postorder (reverse postorder) visit of the whole graph, applying the transfer function at each node to propagate the state to its predecessors (successors).
(2) If anything changed during the last visit, then revisit the whole graph.

Using postorder for backwards analysis and reverse postorder for forwards analysis is extremely important [Aho and Ullman 1977]. For example, postorder traversal visits successors of ϕ before a node ϕ. Thus in a backwards analysis, a postorder traversal will (in the absence of cycles) require only a single pass through the ACFG to reach a fixpoint.

Fig. 12 shows the times to perform correlation inference using the various strate-

gies. A timeout indicates a run that did not terminate within one hour. We found that for our benchmarks, the Queue, Stack, and Double Stack strategies take roughly the same time, and the Postorder Set strategy is slightly faster. Surprisingly, we discovered that getting rid of the worklist altogether is the optimal strategy, performing significantly better on the larger benchmarks (the Linux kernel drivers). For example, on slip, all four worklist algorithms time out after one hour, whereas the no worklist strategy terminates in 17 seconds.

One would expect that the worklist strategies would visit far fewer nodes than the no-worklist strategy, and indeed this is the case for our benchmarks. However, LOCKSMITH uses hashconsing of state data structures to memoize transfer functions on control flow nodes, and thus avoids re-computing the output state for every control flow node visited, if the input has not changed. As a result, the cost of maintaining the worklist far exceeds the cost of redundant visits to nodes. The last column also shows the number of visits of the whole graph, in parentheses.

We do not present the respective measurements for the lock state analysis. We found that because each benchmark includes a relatively small set of locks, the sets of held locks at each program point are small. This makes the lock state analysis quite fast, as little information needs to be propagated. Indeed, for all the benchmark programs the running time for lock state analysis is negligible compared to the total running time, and all five strategies work equally well.

4. SHARING ANALYSIS

As we discussed in Section 3.3.2, during correlation inference we can safely ignore accesses to thread-local data, since such data need not be protected by locks. In this section we show how we compute the set of thread-shared locations. We found that our analysis allows LOCKSMITH to ignore a significant—usually dominant—fraction of the accesses in the program as thread-local.

A simple, but coarse sharing analysis might work by determining the memory locations read or written by each thread in the program; any location that is ever accessed by more than one thread is potentially shared. Our core analysis (Section 4.1) improves on this simple approach by ordering memory accesses according to whether they happen before or after forking a child thread. In particular, even if two threads access the same location, the location cannot be accessed simultaneously, and will not be considered shared by the core analysis, if all accesses in one thread occur before all accesses in the other.

We refine this basic approach in two main ways. First, we use a simple scoping refinement to avoid confusing non-conflicting accesses to local variables with the same names but allocated on the stacks of distinct threads. Without this refinement, if a parent forks two threads that execute the same function, accesses by each thread to its own local variables would be considered conflicting, since two unordered threads would be accessing the "same" location. Second, the core analysis only determines whether there are *any* unordered accesses of some location, but fails to distinguish those accesses that are unordered from those that are not. To gain back some of this lost precision, we use a simple *uniqueness analysis* to track a location from the time it is allocated to the point it becomes assigned to a global variable or passed to a function, both which are events that could lead to

$$\textsc{Eff-Deref}\quad \frac{\Phi_1;\Gamma \vdash e : \mathit{ref}^{\,\rho}(\tau) \qquad \Phi_2^{\varepsilon} = \mathit{Flow}(C,\rho) \qquad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi}{\Phi;\Gamma \vdash\, !\, e : \tau}$$

$$\textsc{XFlow-Ctxt}\quad \frac{\Phi_1 = [\varepsilon_1;(\varepsilon_2 \cup \omega_2)] \qquad \Phi_2 = [\varepsilon_2;\omega_2] \qquad \Phi = [(\varepsilon_1 \cup \varepsilon_2);\omega_2]}{\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi}$$

$$\textsc{Eff-Fork}\quad \frac{\Phi_e;\Gamma \vdash e : \tau \qquad \Phi_e^{\varepsilon} \subseteq \Phi^{\varepsilon} \qquad \Phi_e^{\varepsilon} \cap \Phi^{\omega} \subseteq \mathit{SharLocs}}{\Phi;\Gamma \vdash \mathsf{fork}\ e : \tau}$$

Fig. 13. Contextual effect rules for finding shared locations (selected)

the location being thread-shared—all accesses that occur before those points must be thread-local. Both of these refinements add useful precision at low cost (Section 4.2). We also tried tracking whether an access to an eventually shared location occurs before or after the fork that precipates its being shared, but we found that doing so did not add much benefit beyond the first two refinements (Section 4.3).

4.1 Contextual effects for finding shared locations

Locksmith uses an effect system to infer which locations are thread-shared. Given some expression e, the *effect* ε of e is the set of all locations ρ that e could dereference during its evaluation [Talpin and Jouvelot 1994]. In recent work, we proposed a generalization of effects that we call *contextual effects* [Neamtiu et al. 2008]. The contextual effect of e consists not only of the effect of e's computation, but also the effect α of the computation that has already occurred—called the *prior effect*—and the effect ω of the computation yet to take place—called the *future effect*. At every occurrence of $\mathsf{fork}\ e$ in the program, we compute the effect ε of e, the child thread, and the future effect ω of the parent thread. We consider as thread-shared those locations in the intersection of these two sets. The implementation by default differentiates between read and write effects, and does not consider memory that is only read in parallel to be shared.

Fig. 13 contains selected typing rules from our contextual effect system as applied to this sharing analysis. Full details can be found in our prior paper [Neamtiu et al. 2008]. In these rules, a contextual effect Φ consists of a pair $[\varepsilon;\omega]$, where the first element is the standard effect, and the second element is the future effect.[3] In our implementation, we generate effect-related constraints along with label flow constraints. For simplicity, we present effect typing rules here as a separate judgment, but it would be straightforward to merge these rules with those in Fig. 8.

Eff-Deref types a dereference $!\,e$. In this case, the contextual effect Φ of the entire expression is computed by combining the effect Φ_1 of the subexpression e, defined in the first premise, with the effect Φ_2 of the dereference itself, defined in the second premise. The second premise of Eff-Deref requires that the standard effect of Φ_2 includes ρ and locations that flow to it according to the label flow analysis (written $\mathit{Flow}(C,\rho)$, where the constraint set C is due to the label flow analysis discussed in the previous section). Here the syntax Φ^{ε} refers to the ε (i.e.,

[3] We elide prior effects in these rules, because they are not needed in our sharing computation.

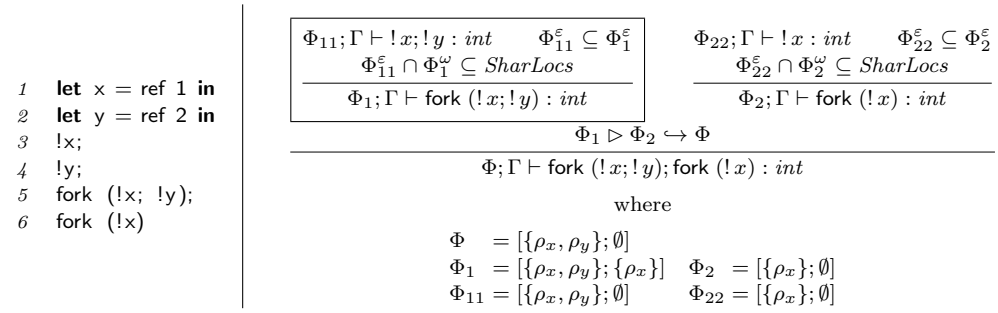


Fig. 14. Example program to illustrate sharing analysis

the first) component of Φ. Given Φ_1 and Φ_2, the combined effect is computed in the last premise by the judgment $\Phi_1 \triangleright \Phi_2 \hookrightarrow \Phi$, defined by rule XFlow-Ctxt.

Looking at XFlow-Ctxt, the first premise requires that because effect Φ_1 occurs *before* Φ_2, the future effect of Φ_1 should include Φ_2's standard effect ε_2 and its future effect ω_2. Intuitively, rule XFlow-Ctxt computes the effect Φ of the sequential composition of Φ_1 and Φ_2. Thus, returning to Eff-Deref, the future effect of Φ_1 naturally must contain $\mathit{Flow}(C, \rho)$, according to XFlow-Ctxt, since the evaluation of e occurs before the dereference. The third premise of XFlow-Ctxt states that the combined effect Φ should contain the standard effects of both Φ_1 and Φ_2. Again, returning to Eff-Deref, we can see that the ε component of Φ will thus contain the effect of e (i.e., Φ_1^ε) and the effect of the dereference itself (Φ_2^ε).

Finally, Eff-Fork type checks thread creation. The second premise of Fork indicates that the standard effect Φ_e^ε of the thread itself should be contained in the effect of the parent. Notice the future effect Φ_e^ω of the thread is unconstrained (effectively making it $\emptyset$). In particular, as expected, it contains no information about the effect of the parent, since the two will execute in parallel. Finally, the third premise adds to the set *SharLocs* the locations that the parent and child thread (and threads they fork) could access in parallel: the intersection of the standard effect of the child thread Φ_e^ε and the future effect Φ^ω of the parent.

We can see this analysis in action in Fig. 14. The left side of the figure shows a simple code example, and the right side shows the typing derivation for the last part of the example, involving the two thread forks.[4] Within this derivation, the boxed portion shows the subderivation for the expression fork $(!\,x; !\,y)$. Here we can see that the standard effect of this expression is $\Phi_1^\varepsilon = \{\rho_x, \rho_y\}$, i.e., locations corresponding to x and y. The future effect Φ_1^ω consists of the location ρ_x—this is because the effect of the subsequent thread spawn is $\{\rho_x\}$, and by EffDeref, this effect is also attributed to the parent thread. Consequently, the intersection of these two effects is $\{\rho_x\}$, indicating that x is potentially accessed simultaneously by two threads. The second thread spawn yields no additional shared locations (since $\Phi_2^\omega = \emptyset$).

There are two interesting things to notice about this example. First, y is accessed

[4]Note that the syntax $e_1; e_2$ can be treated as shorthand for $(\lambda x\!:\!t.e_2)\ e_1$ for some x that does not occur free in e_2, and t being the source type of e_1.

	Pointers			Allocation Sites		
Benchmark	Total	Shared	In scope	Total	Shared	In scope
aget	1,411	258	235	352	64	62
ctrace	1,089	129	116	311	12	12
engine	1,441	60	17	410	11	7
knot	1,238	338	238	321	30	15
pfscan	987	53	48	240	8	7
smtprc	4,275	196	67	1,079	74	46
3c501	10,020	954	913	408	20	20
eql	4,572	2,377	2,168	273	43	35
hp100	19,401	5,268	5,210	497	15	15
plip	13,249	2,867	2,823	466	49	49
sis900	38,624	2,648	2,594	779	11	9
slip	13,748	1,338	1,281	382	20	19
sundance	34,142	3,313	3,267	753	9	9
synclink	51,147	11,621	11,472	1,298	155	139
wavelan	18,799	2,535	2,125	695	128	10
Total	214,143	33,955	32,574	8,264	649	454

Fig. 15. Precision of sharing analysis and scoping

in the parent thread, at line 4, and then subsequently in the first child thread. Nevertheless, our analysis does not consider y as shared, and indeed, y can never be simultaneously accessed by both threads. Second, the example illustrates that it is crucial to include the effect of a child thread in the effect of its parent. Otherwise, we would not have discovered that the two child threads both might simultaneously access x.

Fig. 15 measures the precision of the sharing analysis. For each benchmark, the second column shows the total number of pointers and the third column shows the number of shared pointers, computed by intersecting the effects at thread creation points. We ignore the fourth column for now and return to it in Section 4.2. We also report the results of the sharing analysis in terms of allocation sites, where an allocation site is either a call to malloc() or the location of a variable definition (fifth and sixth columns, ignore the last column). The results underline the effectiveness of using contextual effects to compute shared memory locations; only 16% of all pointers and only 8% of all allocation sites are in the *SharLocs* set computed by Fork. This precision is critical to reducing false positives in Locksmith: thread-local data is almost always accessed without a lock held, and thus if Locksmith incorrectly determines a location is thread-shared, it will likely report a data race for that location.

4.2 Scoping and Uniqueness

We used two additional optimizations to improve the results of the sharing analysis even further. Consider the program in Fig. 16. Here the reference g (line 1) is visible within the function f, which allocates two references x and y (lines 3–4), then writes to them (lines 5–6), and then assigns x to g (line 7). The program calls function f twice (lines 9–10) in two parallel threads. In this example, the sharing analysis from Sections 4.1 and 4.3 will determine that x and y are thread-shared at the writes on lines 5–6, because they are in the effects of both threads. However, in both cases this is overly conservative.

```
1   let g = ref (ref 0) in
2   let f () =
3     let x = (ref 42) in
4     let y = (ref 0) in
5     y := 1;
6     x := 43;
7     g := x
8   in
9   fork f();
10  f()
```

Fig. 16. Limitations of core sharing analysis

Scoping Optimization. Notice that the location y refers to is allocated in the scope of f and never escapes. Hence, the uses of y by the two different threads must refer to distinct memory locations. More generally, when computing thread-shared locations, we can *hide* effects on locations that must be thread-local due to scoping. Formally, we change the type rule for fork as follows:

Eff-Fork-Down

$$\frac{\Phi_e; \Gamma \vdash e : \tau \qquad \Phi_e^{\varepsilon} \subseteq \Phi^{\varepsilon} \qquad \vec{\rho} = \bigcup_{\rho \in fl(\Gamma)} \mathit{Flow}(C, \rho) \qquad \Phi_e^{\varepsilon} \cap \Phi^{\omega} \cap \vec{\rho} \subseteq \mathit{SharLocs}}{\Phi; \Gamma \vdash \mathsf{fork}\ e : \tau}$$

Here we compute the set $\vec{\rho}$ of labels that are reachable in the label flow graph from locations in Γ, meaning they are visible to both the parent and child thread. Then we only add locations to *SharLocs* if they are also in $\vec{\rho}$.

Fig. 15 shows the benefit of this optimization. The fourth column shows the number of shared pointers and the last column shows the number of shared allocation sites when using the revised rule Eff-Fork-Down. The average percentage of shared pointers and allocation sites improves to 15% and 6%, respectively.

Uniqueness Analysis. Returning to the example program in Fig.16, note that the scoping optimization does not apply to x, since it escapes via a write to the global variable g. However, while x does escape the scope of f eventually, there is no way that it can be accessed by any other thread during the write at line 6, since the aliasing on line 7 has not yet occurred. So, we can safely ignore the access to x at line 6, not requiring it to be protected by a lock.

This situation can occur in C programs when a struct is malloc'd and initialized thread-locally before becoming shared. To model this situation precisely, we developed a *uniqueness analysis* to determine when a memory access is guaranteed to be thread-local because the accessed location has not yet become aliased. We then ignore these accesses during the correlation analysis.

Our uniqueness analysis is implemented as a simple, intraprocedural dataflow analysis. Whenever a location is created (through a local variable definition or a call to malloc), we mark it as unique. When a unique location ρ is assigned to any non-unique location, or a variable with location ρ has its address taken, ρ becomes non-unique. Using this analysis, we discover that x is unique on line 6 in

Benchmark	With scoping and uniqueness		Without scoping and uniqueness	
	Time (s)	Warnings	Time (s)	Warnings
aget	0.85	62	0.88	64
ctrace	0.59	10	0.58	10
engine	0.88	7	0.69	11
knot	0.78	12	0.83	28
pfscan	0.46	6	0.43	7
smtprc	5.37	46	5.46	74
3c501	9.18	15	8.93	15
eql	21.38	35	20.01	35
hp100	143.23	14	140.97	14
plip	19.14	42	18.75	44
sis900	71.03	6	70.77	6
slip	16.99	3	16.65	3
sundance	106.79	5	103.21	5
synclink	1521.07	139	1454.91	155
wavelan	19.70	10	20.60	128

Fig. 17. Scoping and uniqueness

our example, and hence is thread-local.

Fig. 17 shows the aggregate effect of these two optimizations for LOCKSMITH. We compare the running time and number of warnings when using both techniques, shown in the second and third columns, against the running time and number of warnings without them, in the fourth and fifth columns. Our results show that the effect on running times is negligible, but in several of the benchmarks the two optimizations determine that many accesses are thread-local, significantly reducing the number of warnings. In all benchmarks but ctrace, the gain in precision is due purely to the scoping optimization rather than the uniqueness analysis.

4.3 Flow-sensitive sharedness

Consider the example in Fig. 14 again. Suppose we add acquire l and release l before and after, respectively, each access $!x$ in the two child threads (lines 5 and 6), for some lock l. Also suppose that x is aliased by a global variable immediately after its allocation. This changed program will be correct, but our analysis will falsely complain that the dereference of x at line 3 is a potential race because it is not protected by a lock. Moreover, the scoping optimization and uniqueness analysis described in Section 4.2 do not apply, as x is aliased by a global variable. However, at this dereference site, x is not actually shared, and thus requires no guarding lock. This is because it will not become shared until both of the child threads are spawned, at lines 5 and 6.

In this case, as in the uniqueness analysis in Section 4.2, we can safely ignore a memory access when the accessed location is thread-local during the access, even if it later becomes shared.

To address this problem, we can use a simple dataflow analysis along the ACFG to determine at which sites in the program a location can be dereferenced after it becomes shared. Any sites that occur before it becomes shared can be dropped from correlation inference. The transfer functions for the analysis are straightforward, as shown in Fig. 18. In essence, we seed the analysis at the fork points with those

ϕ kind	$Sh_{out}(\phi)$
Fork	the portion of *SharLocs* computed at this fork site
all others	$Sh_{in}(\phi)$

Fig. 18. Sharedness inference

Benchmark	No dataflow		Context-insensitive		Context-sensitive	
	Time(s)	Warnings	Time(s)	Warnings	Time(s)	Warnings
aget	0.80	62	0.85	62	1.73	62
ctrace	0.58	10	0.59	10	0.81	10
engine	0.89	7	0.88	7	1.27	7
knot	0.64	12	0.78	12	1.70	12
pfscan	0.45	6	0.46	6	0.57	2
smtprc	5.11	46	5.37	46	16.71	46
3c501	9.29	15	9.18	15	11.46	15
eql	21.39	35	21.38	35	24.35	35
hp100	143.45	14	143.23	14	172.97	14
plip	19.18	42	19.14	42	37.80	42
sis900	71.96	6	71.03	6	82.35	6
slip	17.05	3	16.99	3	18.92	3
sundance	106.89	5	106.79	5	117.01	5
synclink	1,513.94	139	1,521.07	139	1,823.91	139
wavelan	19.69	10	19.70	10	26.84	10
Total	1,931.31	412	1,937.44	412	2,338.40	408

Fig. 19. The effect on LOCKSMITH's results of different dataflow strategies for finding shared location dereference sites

locations made possibly shared due to that fork. Specifically, we combine the type rules in Fig. 13 with Fig. 7 to get a judgement of the form $C; \phi; \Phi; \Gamma \vdash \mathsf{fork}\ e : \tau; \phi$ for typing fork e expressions. Then, at every such point in the program, we set $\Phi_e^\varepsilon \cap \Phi^\omega \subseteq Sh_{in}(\phi)$ to seed the analysis.

Unfortunately, while this optimization adds precision in general, it is not very helpful for our benchmarks, as shown in Fig. 19. Columns 2 and 3 in the figure show the results of LOCKSMITH when using the contextual effects analysis (including scoping and uniqueness) to compute shared locations, without using the dataflow analysis, while columns 4 and 5 include the dataflow analysis. We see that the running times are nearly the same, but unfortunately, so are the warning counts. One reason for this is that a location that is eventually shared may be written to by the parent *after* the child is forked, and then shared with the child by writing to a global variable. The dataflow analysis conservatively considers all accesses after the fork to be potentially shared. When we make the dataflow analysis context-sensitive (Section 6.4), we see an improvement in one case—pfscan has 4 fewer warnings. However the context-sensitive results are clearly more expensive to compute. Thus, because employing dataflow is essentially free and could increase precision, we enable it by default, while context-sensitive dataflow for discovering sharing can be enabled via a command-line flag. (Thus the results from columns 4 and 5 in Fig. 19 match the results in Fig. 4.)

	Field-sensitive					Field-insensitive				
Program	CGen Tm (s)	Total Tm (s)	Labels	Shr	Wrn	CGen Tm (s)	Total Tm (s)	Labels	Shr	Wrn
aget	0.55	0.85	5,634	62	62	0.50	0.67	5,490	62	62(11)
ctrace	0.40	0.59	4,351	12	10	0.38	0.53	4,285	15	13(5)
engine	0.76	0.88	5,051	7	7	0.79	0.91	4,989	59	59(7)
knot	0.55	0.78	4,752	15	12	0.52	0.83	4,566	24	21(12)
pfscan	0.36	0.46	4,143	7	6	0.36	0.46	4,139	15	14(5)
smtprc	3.09	5.37	14,815	46	46	3.08	5.14	14,917	97	97(43)
3c501	7.92	9.18	25,905	20	15	7.60	18.56	22,976	42	42(6)
eql	2.72	21.38	8,954	35	35	2.39	17.99	7,484	42	42(18)
hp100	35.92	143.23	31,609	15	14	34.18	976.12	22,214	41	41(10)
plip	16.41	19.14	24,124	49	42	17.82	103.21	18,969	60	60(6)
sis900	65.66	71.03	84,797	9	6	60.45	132.18	71,630	42	42(6)
slip	15.11	16.99	25,371	19	3	15.44	33.24	18,333	56	31(5)
sundance	96.72	106.79	73,552	9	5	81.44	6835.26	61,540	44	44(8)
synclink	1433.56	1521.07	68,643	139	139	1232.05	timeout	n/a	171	n/a
wavelan	17.89	19.70	30,052	10	10	16.90	40.19	21,071	43	44(6)

Fig. 20. Field sensitivity

5. EFFICIENTLY AND PRECISELY MODELING STRUCT AND VOID* TYPES

In this section we discuss some additional techniques that we used to increase the speed and precision of LOCKSMITH as applied to C programs. In particular, we explain how we analyze struct and void* types effectively. We initially explored some of these ideas when developing CQual [Foster et al. 2006]. This paper's contribution is to express the ideas more precisely, in particular using a new formalism for our analysis of structures, and to measure their costs and benefits directly, measuring LOCKSMITH's performance and precision when using one or the other of several different strategies.

5.1 Field sensitivity

In designing a static analysis for C, one important decision is whether to model C struct types *field-insensitively* or *field-sensitively* [Heintze and Tardieu 2001]. In a field-insensitive analysis, all fields of a struct type are conflated, i.e., x.f and x.g are treated as the same location by the analysis for any fields f and g. In a field-sensitive analysis, different struct fields are distinguished, i.e., x.f and x.g are treated as different locations.[5] These two design points potentially trade off efficiency and precision—field-insensitive analysis may be less precise but more scalable, because it distinguishes fewer locations. In particular, if there are m occurrences of struct types, each of which has n fields, then field-sensitive analysis would annotate $O(mn)$ types with fresh locations, whereas field-insensitive analysis would only annotate $O(m)$ types.

We implemented support for both field-insensitive and field-sensitive analysis in LOCKSMITH. Field insensitivity is actually somewhat tricky to use in an analysis like LOCKSMITH, which is layered on top of C types: True field insensitivity would throw away those types, thereby requiring some significant approximations in the analysis (e.g., conflating all labels of all fields of a struct). Thus, since we had a

[5]There is a third design point, a *field-based* analysis, in which x.f and x.g are different, but x.f and y.f are the same if x and y are instances of the same struct type. We did not explore this option for LOCKSMITH, however, because it would greatly reduce the effectiveness of context sensitivity, which we found was important to improving precision.

full field-sensitive analysis, we opted to implement a simple variation to simulate the following key aspect of field insensitivity: each instance of a struct uses a single location ρ to represent the top-level location of all fields. Otherwise we use the standard flow-sensitive implementation. For example, suppose we have a struct with three fields int x, int $*$ y, and int $*$ z. Then for an instance of this struct, fields x, y, and z would have types $ref^{\rho}(int)$ (since x can be written to, it has a *ref* type), $ref^{\rho}(ref^{\rho'_y}(int))$, and $ref^{\rho}(ref^{\rho'_z}(int))$, respectively.

Fig. 20 compares the two approaches. For each style of analysis, we list the time for constraint generation (including annotating types with fresh labels, i.e., abstract locations and locks), the total analysis time, the number of generated labels (locations and locks), the number of shared locations, and the number of reported warnings. Note that since LOCKSMITH issues one warning per unprotected shared location, this means warning counts from field-insensitive and field-sensitive analysis are incomparable: A single warning from field-insensitive analysis might actually correspond to multiple races from the field-sensitive analysis. To normalize the warning counts, if there is a warning on a location corresponding to a struct field, we count that as n warnings for comparison purposes, where n is the number of fields in that struct instance, as computed by our lazy fields algorithm (Section 5.2). The normalized warnings for the field-insensitive analysis are listed in the rightmost column, with the raw number of warnings in parentheses.

These results show that the field-insensitive analysis takes less time to generate constraints and generally creates fewer labels than the field-sensitive analysis.[6] However, even for the very slightly reduced precision with field insensitivity—conflating the locations of all struct fields—many more locations are considered shared, which in turn makes LOCKSMITH as a whole both less precise, as evidenced by the warning counts, and considerably slower, since it must infer correlations for more aliases.

5.2 Lazy struct fields

Since field-sensitive analysis can potentially be expensive, in order to achieve the performance reported in Fig. 20 we had to implement field sensitivity carefully. At first, we used a naive approach, in which we fully annotated all field types of all struct instances. We quickly ran into scalability problems, however, and were not able to analyze any but the smallest benchmarks.

Examining our benchmarks, we found that many C struct types have a large number of fields (up to 300!). However, many large struct types are declared by a library and only used in a small subset of the code, and this subset often accesses only a fraction of the struct's total fields. Our naive implementation was assigning abstract locations and locks to all of the rarely- or never-used fields, wasting memory and time generating constraints among them.

To regain scalability, our field-sensitive implementation *lazily* annotates the fields

[6] For one program, smtprc, there are fewer field-sensitive labels than field-insensitive labels. This is because the field-insensitive analysis always creates at least one label (for the location of all fields) for every occurrence of a struct, whereas the field-sensitive analysis might avoid creating any labels for a struct instance if it is created and then immediately equated with another struct instance.

$$\begin{array}{rcl} \tau & ::= & \ldots \mid t \times^{\zeta} t \\ C & ::= & \ldots \mid \zeta[j] = \tau \mid \zeta = \zeta \end{array}$$

$$\zeta \in \textit{pair labels}$$

$$\begin{array}{rcl} \langle\langle t_1 \times t_2 \rangle\rangle & = & t_1 \times^{\zeta} t_2 \qquad \zeta \textit{ fresh} \\ \langle\langle \textit{other } t \rangle\rangle & = & \tau \text{ as in Fig. 7(a)} \\ \langle\langle \tau \rangle\rangle & = & \tau' \text{ where } \tau' \text{ is } \tau \text{ with fresh } \rho, \ell, \phi, \zeta\text{'s} \end{array}$$

(a) Auxiliary definitions

PAIR

$$\frac{C; \phi; \Gamma \vdash e_1 : \tau_1; \phi_1 \qquad C; \phi_1; \Gamma \vdash e_2 : \tau_2; \phi_2 \qquad \zeta \textit{ fresh} \qquad C \vdash \zeta[1] = \tau_1 \qquad C \vdash \zeta[2] = \tau_2 \qquad t_j = \textit{std type of } e_j,\ j \in 1..2}{C; \phi; \Gamma \vdash (e_1, e_2) : t_1 \times^{\zeta} t_2; \phi_2}$$

PROJ

$$\frac{C; \phi; \Gamma \vdash e : t_1 \times^{\zeta} t_2; \phi' \qquad j = 1, 2 \qquad \tau_j = \langle\langle t_j \rangle\rangle \qquad C \vdash \zeta[j] = \tau_j}{C; \phi; \Gamma \vdash e.j : \tau_j; \phi'}$$

(b) Type inference rules

$$\begin{array}{rcl} C \cup \{t_1 \times^{\zeta} t_2 \leq t_1 \times^{\zeta'} t_2\} & \Rightarrow & C \cup \{\zeta = \zeta'\} \\ C \cup \{\zeta = \zeta'\} & \Rightarrow & C[\zeta/\zeta'] \\ C \cup \{\zeta[j] = \tau_1, \zeta[j] = \tau_2\} & \cup \Rightarrow & \{\tau_1 = \tau_2\} \end{array}$$

(c) Constraint resolution rule

Fig. 21. Type inference rules for modeling pairs lazily

of struct types [Foster et al. 2006]. Initially we leave all occurrence of struct types unannotated. Then whenever we encounter a field access in the program, we add the accessed field to the corresponding struct type. If we create a label flow constraint between two struct types, we equate the labels on their fields.

Fig. 21 extends the inference rules from Fig. 7 to implement this lazy field generation algorithm. Our formalism uses pairs instead of general structs, and so we illustrate our approach by modeling pairs lazily. Fig. 21(a) gives the new type and constraint definitions. Types are the same as before, except pair types now have the form $t \times^{\zeta} t$, where ζ is a *pair label*. Notice that this pair type contains unannotated types t. The pair label ζ is used to track the labeled components of the type: the constraint $\zeta[j] = \tau$ indicates that component j of any pair type annotated with ζ has annotated type τ. The constraint $\zeta_1 = \zeta_2$ indicates the corresponding components of pairs labeled with ζ_1 and ζ_2 have the same types.

We also extend $\langle\langle \cdot \rangle\rangle$ to introduce fresh pair labels. As shown, when we translate a standard pair type into a labeled pair type, we tag it with a fresh pair label but do *not* introduce labels for the component types. This annotation function is used in LAM from Fig. 7 to give fresh labels to programmer-supplied types. Thus we see the laziness of this approach: we do not automatically create labels for a pair type

when it is mentioned in the program text.

Fig. 21(b) gives our modified type rules. PAIR types pair creation, which now associates a fresh pair label ζ with the output type and constrains the components of ζ to their corresponding labeled types. Notice that there is no laziness in this rule, because we have labeled types for e_1 and e_2. Here we compute the standard types t_1 and t_2 (of e_1 and e_2, respectively) by stripping off all labels from τ_1 and τ_2. Using standard types keeps the analysis "lazy." Standard types have no (wasted) location or lock annotations; instead, we track the omitted locations off to the side using ζ.

More interestingly, PROJ types projection, which lazily annotates only the accessed component of the pair. We create a type τ_j with fresh labels and constrain it to be equal to $\zeta[j]$. In our implementation, rather that creating constraints $\zeta[j] = \tau$ and then solving them later, we solve these constraints on-line, as we apply the type inference rules. We maintain a partial mapping ST from each $\zeta[j]$ to its type. Initially ST is empty. In PROJ, if $ST(\zeta[j])$ exists, we use it in place of τ_j rather than making up a fresh type. Otherwise, we do make a fresh type τ_j and set $ST(\zeta[j]) = \tau_j$. Our implementation for PAIR is similar.

Fig. 21(c) gives the resolution rules for our new constraint forms. The first rewriting rule is for subtyping among pair types. Here we assume the standard types of the pairs match (i.e., the program passes the standard type checker) and equate the pair labels, which are merged by the next rewriting rule. The last rule equates types for different occurrences of $\zeta[j]$.

Notice that we lose some precision here compared to the previous type system. In our lazy approach, subtyping among pair types requires their component types to be equal. The constraint resolution rule from Fig. 8, on the other hand, permits subtyping the component types, which is more precise. However, in practice, C programs mostly manipulate pointers to structs, and subtyping pointer types requires that the pointed-to types are equal (Fig. 8), which negates any benefits of the more precise subtyping rule. Thus we lose little practical precision with this approach.

Moreover, in our implementation, we maintain pair labels ζ in a union-find data structure. Given the constraint $\zeta = \zeta'$, we unify the two sides of the equation and equate the associated types in ST. This unification process reduces the need to create types for fields. For example, if $\zeta[0] = \tau$ and $\zeta'[1] = \tau'$ are the only mappings in ST, then after we unify the pair labels we will have $\zeta[0] = \zeta'[0] = \tau$ and $\zeta[1] = \zeta'[1] = \tau'$, without creating any additional field types. Thus, we also gain efficiency from this approach.

We already saw in Fig. 20 that field sensitivity, while it typically produces more labels that field-insensitive analysis, does not yield an inordinate number of labels. Fig. 22 gives some measurements that illustrate why this is the case. For each benchmark, we list the total number of struct types in the program, the number of instances of all struct types, the total possible number of instance fields, and the instance fields that are actually used, both in absolute numbers and as a percentage of the total number of fields. We define the total number of instance fields as all the used fields of all instances of struct types. For example, if a program defines two instances of a single struct type with three fields and the program accesses

Benchmark	struct Types	Instances		
		Total	Total fields	Used fields
aget	11	61	563	179 (32%)
ctrace	12	43	427	79 (19%)
engine	14	48	618	85 (14%)
knot	16	68	565	156 (28%)
pfscan	13	65	639	78 (12%)
smtprc	19	80	1,019	106 (10%)
3c501	37	1,025	14,691	3,768 (26%)
eql	33	888	9,617	2,301 (24%)
hp100	36	1,786	41,537	12,072 (29%)
plip	46	1,986	24,161	7,320 (30%)
sundance	58	4,141	51,400	16,504 (32%)
sis900	60	4,511	58,952	18,106 (31%)
slip	39	1,426	31,529	8,319 (26%)
synclink	49	5,431	68,423	38,497 (56%)
wavelan	58	2,879	31,823	11,608 (36%)
Total	501	24,438	335,964	119,178 (35%)

Fig. 22. Lazy field statistics

all fields of one instance and two of the other so that the lazy field analysis only populates those with location labels, we "use" five instance fields out of a total of six. This data shows that on average across all the benchmarks, only 35% of the possible instance fields are actually used. Thus, lazy field analysis is effective because modeling those fields would otherwise consume memory and time with no gain in precision.

5.3 Modelling `void*`

In addition to deciding how to model structs, another important decision in analyzing C code is determining how to model `void*`, which is typically used by C programmers to express polymorphism. For LOCKSMITH, the key choice is how to track the abstract locations and locks of types that "flow" to or from `void*` positions. We experimented with three different strategies:

—*Conflate `void*`.* Since any type might be cast to or from `void*`, an imprecise but sound approach is to conflate all abstract locations and locks that reach a `void*` type. More precisely, let $\tau = ref^{\rho}(\texttt{void})$ be an occurrence of `void*` in the program, labeled with location ρ. If we ever derive a constraint $\tau \leq \tau'$ or $\tau' \leq \tau$, we equate all the locations in τ' with ρ. This is quite conservative, since it effectively aliases all locations reachable from a type that flows to or from a `void*`. If any locks occur inside τ', LOCKSMITH warns about the loss of precision, and considers these locks non-linear and thus unable to protect memory locations.

—*Singleton `void*`.* In the previous approach, we conflated labels because `void*` types may be cast arbitrarily. However, it could be that a particular `void*` in the program is used with only one concrete type. We thus tried refining the previous approach as follows. Let $\tau = ref^{\rho}(\texttt{void})$ be an occurrence of `void*`. We wish to define the partial function *base_type* as a map from `void*` occurrences to the single concrete type it could be replaced with. Given a constraint $\tau \leq \tau'$ or $\tau' \leq \tau$, there are three cases. If $base\_type(\tau)$ is as yet undefined, we set

	Conflate `void*`		Single `void*`		Type-based `void*`	
Benchmark	Tm (s)	Wrn	Tm (s)	Wrn	Tm (s)	Wrn
aget	1.89	72	1.98	72	0.85	62
ctrace	0.60	10	0.61	10	0.59	10
engine	0.90	7	0.89	7	0.88	7
knot	1.46	12	0.77	12	0.78	12
pfscan	0.45	6	0.46	6	0.46	6
smtprc	5.22	46	5.26	46	5.37	46
3c501	246.24	121	358.98	121	9.18	15
eql	12.40	41	12.58	42	21.38	35
hp100	105.80	50	782.52	50	143.23	14
plip	413.96	60	4,011.22	148	19.14	42
sis900	778.20	149	6,037.00	152	71.03	6
slip	88.79	68	timeout	n/a	16.99	3
sundance	3,188.32	148	7,661.57	148	106.79	5
synclink	timeout	n/a	timeout	n/a	1,521.07	139
wavelan	168.28	112	169.03	111	19.70	10

Fig. 23. Performance of `void*` strategies

it to τ'. Otherwise, if τ' has the same shape (i.e., underlying standard type) as $\mathit{base\_type}(\tau)$, we generate the constraint $\mathit{base\_type}(\tau) \leq \tau'$ or $\tau' \leq \mathit{base\_type}(\tau)$, as appropriate. Otherwise, we revert to the above conflation strategy, and collapse $\mathit{base\_type}(\tau)$ and any other types that flow to τ. As before, if we collapse types then we treat any locks occurring inside τ as non-linear, and assume they do not protect any locations. This approach is sound but is more precise than conflation in the case of `void*`s that are used with only one type. Indeed, we found that approximately one third of `void*` pointers in our benchmarks are only cast to or from one non-`void*` type.

—*Type-based `void*`.* Finally, we can improve further on the previous approach if we are willing to sacrifice completely sound modeling of `void*`. For each standard type t that flows to $\tau = \mathit{ref}^{\rho}(\texttt{void})$, we create a type $\mathit{base\_type}(\tau, t)$. Then given a constraint $\tau \leq \tau'$ or $\tau' \leq \tau$, we generate the constraint $\mathit{base\_type}(\tau, t) \leq \tau'$ or $\tau' \leq \mathit{base\_type}(\tau, t)$, where t is the underlying standard type from τ'. Thus, our *base_type* function is now indexed by the shape of the underlying type, similar to a C untagged union. We will never collapse types (or mark locks as non-linear) using this strategy. Modeling `void*`s this way is unsound, since we may miss relationships among different types that are cast to or from `void*`—this would be like storing a pointer into an untagged union but then extracting an integer. Our assumption is that this behavior is unlikely or harmless to our analysis, since if it were not the program would likely fail. By default, LOCKSMITH uses this strategy, possibly sacrificing soundness, to reduce the number of false positives. The user can switch to one of the above alternative, conservative strategies using a run-time flag.

For all of these approaches, we need to integrate our modeling of `void*` with our lazy modeling of structs. That is, we might discover during constraint resolution that a `void*` type points to a struct type, or that a struct type contains a `void*` type.

```
1   struct cache_entry {
2     int refs;
3     pthread_mutex_t refs_mutex;
4     ...
5   };
6
7   void cache_entry_addref(cache_entry *entry) {
8     ...
9     pthread_mutex_lock(&entry→refs_mutex);
10    entry→refs++;
11    pthread_mutex_unlock(&entry→refs_mutex);
12    ...
13  }
```

Fig. 24. Example code with a per-element lock

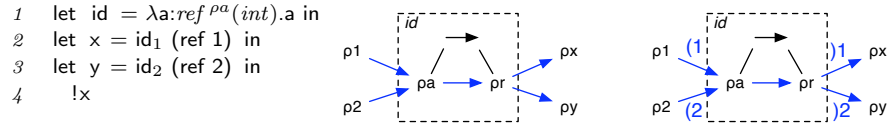
Fig. 23 compares the running times and number of warnings produced for each `void*` strategy. We can see that on most of the benchmarks, the type-based `void*` approach yields both many fewer warnings and is much faster than the other approaches. This phenomenon is similar to that of Fig. 20: the more precise analysis causes fewer locations to be conflated, which both speeds up the computation of the *Flow*() sets and reduces the number of shared locations. Moreover, the type-based warnings are much easier to follow for the user, as there is much less false aliasing due to conflation. Though the type-based approach could be unsound, a manual analysis of a sample of the additional warnings produced by the alternative analyses found no additional races.

6. CONTEXT SENSITIVITY

So far, the analyses we have presented have been context-insensitive. While the resulting analysis is easy to understand and implement, its precision suffers. This section considers how we add context sensitivity to solve these problems.

We consider two kinds of context sensitivity. The first is standard: we would like to analyze different calls to the same function distinctly, rather than conflate them. We use an approach pioneered by Reps et al. [1995], who showed how to reduce the problem of tracking flow context-sensitively through function calls to the problem of context-free language (CFL) reachability. The insight is to view a call to and return from some function f as a string containing a left and right parenthesis, respectively, subscripted by an index identifying the call site. Thus the problem of tracking flow through function calls is one of matching like-subscripted parentheses. We draw ideas more directly from Rehof and Fähndrich [2001] and Fähndrich et al. [2000], which apply Reps et al.'s idea to label flow analysis and points-to analysis, respectively. To solve context-free language reachability constraints, we use Banshee, which encodes and solves the problem using set constraints.

In addition to function calls, the analyses presented so far conflate all locks in a recursive data structure, causing further imprecision. Fig. 24 illustrates this problem with code extracted from the knot web server. Here cache_entry is a linked list with a per-node lock refs_mutex that guards accesses to the refs field. Without some added context sensitivity, Locksmith conflates all the locks and locations in



(a) Source program (b) Monomorphic analysis (c) CFL polymorphic analysis

Fig. 25. Precision gain for label flow from context-sensitive analysis with universal polymorphism

the data structure. As a result, it does not know exactly which lock is held at the write to entry→refs, and reports that entry→refs may not always be accessed with the same lock held, falsely indicating a potential data race.

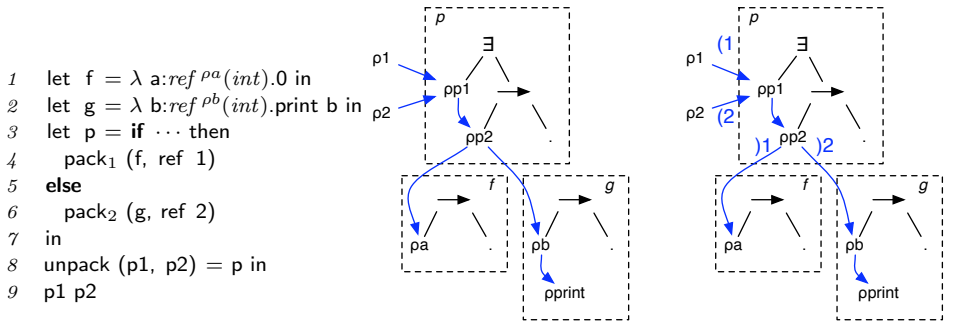
We augment our analysis to solve this problem using a kind of "data structure sensitivity." To do this, we provide support for existential quantification to relate elements within data structures. This support is encoded as a CFL reachability problem in a manner similar to our encoding of context sensitivity for function calls. We add annotations to specify that in type cache_entry, the fields refs and refs_mutex should be given existentially quantified labels. Then we add `pack` annotations when cache_entry is created and `unpack` annotations wherever it is used, e.g., within cache_entry_addref. LOCKSMITH then can determine that the refs_mutex lock in a node always guards the refs field in that node.

6.1 Examples

We provide two examples to illustrate our approach to supporting context sensitivity via CFL reachability, considering both universal and existential quantification.

Universal context sensitivity. Fig. 25 gives the canonical example illustrating the benefits of context sensitivity for label flow analysis. (We discuss context-sensitive dataflow analysis below.) This program defines an identity function id and applies it twice on distinct locations, on lines 2 and 3. As in Section 2, we have indexed each syntactic use of id with an integer. Fig. 25(b) shows a simplification of the constraint graph produced by applying the context-insensitive type rules in Fig. 7. Here ρi is the location containing integer i, locations ρa and ρr are from the domain and range types of id, respectively, and ρx and ρy are from the types of x and y. Notice that when we compute the transitive closure of these constraints, we will discover that both $\rho 1$ and $\rho 2$ flow to ρx, even though only $\rho 1$ may actually reach the dereference of x at run time.

Fig. 25(c) shows how using context-free language reachability, which we discussed briefly in Section 2, eliminates this imprecision. When we use the type of id, we label the generated constraints with indexed parentheses. In our example, the call on line 2 yields edges $\rho 1 \rightarrow^{(1} \rho a$ and $\rho a \rightarrow^{)1} \rho x$, and analogously for the call on line 3. When we resolve the constraints in Fig. 25(c), we only transitively close paths that contain no mismatched edges. In this case, that means there is a path from $\rho 1$ to ρx, since (1 matches)1, but there is no path from $\rho 2$ to ρx, since (1 does not match)2.



(a) Source program (b) Monomorphic analysis (c) CFL polymorphic analysis

Fig. 26. Precision gain for label flow from context-sensitive analysis with existential polymorphism

Existential context sensitivity. Consider the example shown in Figure 26(a). In this program, function g prints its argument, whereas function f does not.[7] In lines 3–7 of this program we create existentially quantified pairs using pack operations in which f is paired two distinct locations, initialized with values 1 and 2. As in the example for universal polymorphism above, we have indexed each syntactic occurrence of pack with an integer. Using an if, we conflate these two pairs, binding one of them to p. In the last line we use p by applying its first component to its second component.

Fig. 26(b) shows a simplification of the constraint graph produced by applying the context-insensitive type rules in Fig. 7. Again, ρi is the location containing integer i, locations ρa and ρb are from the domain types of f and g respectively, and $\rho p1$, $\rho p2$ are from the types of the existential package p, and $\rho print$ is the domain of the print instruction. In this example, f can only be applied to the reference to 1, and g to the reference to 2. However, in the transitive closure of these constraints $\rho 1$ flows to $\rho print$, suggesting a flow that cannot happen at run-time.

Fig. 26(c) shows how using context-free language reachability eliminates this imprecision. When we create the type of p, we label the generated constraints with indexed parentheses. (This is as opposed to the case of universal polymorphism above, where the parenthesis edges are introduced at the point of use rather than the point of creation.) In our example, the pack on line 4 yields edges $\rho 1 \rightarrow^{(1} \rho p1$ and $\rho p2 \rightarrow^{)1} \rho a$, and similarly for the call on line 6. We resolve the constraints in the same way as in the previous example, which means that there is a path from $\rho 2$ to $\rho print$, since (2 matches $)2$, but there is no path from $\rho 1$ to $\rho print$, since (1 does not match $)2$.

In the remainder of this section, we show how to incorporate CFL context sensitivity into our system and also apply it to dataflow analysis (analogously to Reps et al [Reps et al. 1995]). In Locksmith we apply the ideas of universal and existential polymorphism from label flow analysis to correlation inference, in the same way. We focus on universal polymorphism here, and refer the reader to our previ-

[7] For the purpose of this example, we augment the language with a print expression that prints the contents of its argument.

$$
\begin{array}{rcl}
e & ::= & \ldots \mid \mathsf{let}\ f = v\ \mathsf{in}\ e_2 \mid f_i \mid \mathsf{fix}\, f{:}t.v \\
\sigma & ::= & (\forall.\tau, \vec{\eta}) \\
\eta & ::= & \ell \mid \rho \mid \phi \\
C & ::= & \ldots \mid \eta \preceq^i_+ \eta \mid \eta \preceq^i_- \eta
\end{array}
$$

(a) Extensions to source language, types, and constraints

LET

$$\frac{C; \phi_1; \Gamma \vdash v_1 : \tau_1; \phi_2 \qquad \vec{\eta} = \mathit{fl}(\Gamma) \qquad C; \phi_2; \Gamma, f : (\forall.\tau_1, \vec{\eta}) \vdash e_2 : \tau_2; \phi_3}{C; \phi_1; \Gamma \vdash \mathsf{let}\ f = v_1\ \mathsf{in}\ e_2 : \tau_2; \phi_3}$$

INST

$$\frac{\tau' = \langle\langle \tau \rangle\rangle \qquad C \vdash \tau \preceq^i_+ \tau' \qquad C \vdash \vec{\eta} \preceq^i_\pm \vec{\eta}}{C; \phi; \Gamma, f : (\forall.\tau, \vec{\eta}) \vdash f_i : \tau'; \phi}$$

FIX

$$\frac{\tau = \langle\langle t \rangle\rangle \qquad \vec{\eta} = \mathit{fl}(\Gamma) \qquad C; \phi_1; \Gamma, f : (\forall.\tau, \vec{\eta}) \vdash v : \tau'; \phi_2 \qquad C \vdash \tau' \leq \tau \qquad \tau'' = \langle\langle t \rangle\rangle \qquad C \vdash \tau \preceq^i_+ \tau'' \qquad C \vdash \vec{\eta} \preceq^i_\pm \vec{\eta}}{C; \phi_1; \Gamma \vdash \mathsf{fix}\, f{:}t.v : \tau''; \phi_2}$$

(b) Additional type inference rules

Fig. 27. Extensions to Fig. 7 for context sensitivity

ous work on existential context sensitivity [Pratikakis et al. 2006a]. We end with experimental results illustrating the precision benefits of context sensitivity.

6.2 Labeling and Constraint Generation

Fig. 27 extends our core constraint generation rules from Fig. 7, as in Pratikakis et al. [2006a; 2006b]. We begin by introducing three new kinds of expressions, as shown in Fig. 27(a). Expression $\mathsf{let}\ f = v\ \mathsf{in}\ e$ binds f to value v during evaluation of e, assigning f a polymorphic type. Here we assume that the names of polymorphically-typed variables are syntactically distinct from other (monomorphically) typed variables. In practice for C, we only introduce polymorphism for functions, whose names are easily identified. Next, the expression f_i corresponds to a use of variable f annotated with index i. In practice, we simply assign a distinct index to each syntactic use of a function name. Finally, $\mathsf{fix}\, f{:}t.v$ binds f to v recursively inside of v (which will always be a function in practice). In C, all functions are potentially mutually recursive, and so we treat a C program as if it were a set of nested fix bindings. Notice that we do not need to visit a function's definition before its calls, because we can generate the right polymorphic function type from simply the function declaration. We then generate the constraints for the function body later upon encountering the function definition. So, it is not necessary to visit function definitions in a topological order of the call graph.

Let- and fix-bound variables f are assigned polymorphic *type schemes* σ of the form $(\forall.\tau, \vec{\eta})$. Here τ is the generalized type, and $\vec{\eta}$ is the set of labels (i.e., ρ's, ℓ's and ϕ's) that are *not* quantified in the type scheme [Henglein 1993]. During typing of the new language forms, we generate *instantiation constraints* of the form $\eta \preceq^i_p \eta'$, where p is a *polarity*, either $+$ or $-$, and i is an index. Informally, such a constraint means that there is some substitution S_i that instantiates η to η'. The polarity indicates the direction of "flow". More particularly, a constraint $\eta \preceq^i_+ \eta'$

$$
\begin{array}{rcl}
C \cup \{int \preceq^i_p int\} & \Rightarrow & C \\
C \cup \{ref^{\rho}(\tau) \preceq^i_p ref^{\rho'}(\tau')\} & \Rightarrow & C \cup \{\rho \preceq^i_p \rho', \tau \preceq^i_{\pm} \tau'\} \\
C \cup \{lock^{\ell} \preceq^i_p lock^{\ell'}\} & \Rightarrow & C \cup \{\ell \preceq^i_p \ell'\} \\
C \cup \{t_1 \times^{\zeta} t_2 \preceq^i_p t_1 \times^{\zeta'} t_2\} & \Rightarrow & C \cup \{\zeta \preceq^i_{\pm} \zeta'\} \\
C \cup \{(\tau_1, \phi_1) \to (\tau_1', \phi_1') \preceq^i_p (\tau_2, \phi_2) \to (\tau_2', \phi_2')\} \Rightarrow & & \\
\qquad C \cup \{\tau_1 \preceq^i_p \tau_2, \phi_1 \preceq^i_p \phi_2, \tau_1' \preceq^i_p \tau_2', \phi_1' \preceq^i_p \phi_2'\} & & \\
\\
C \cup \{\zeta \preceq^i_{\pm} \zeta'\} \cup \{\zeta[j] = \tau\} & \cup \Rightarrow & \{\tau \preceq^i_{\pm} \zeta'[j]\} \\
C \cup \{\zeta' \preceq^i_{\pm} \zeta\} \cup \{\zeta[j] = \tau\} & \cup \Rightarrow & \{\zeta'[j] \preceq^i_{\pm} \tau\} \\
\\
C \cup \{\rho_1 \preceq^i_- \rho_0, \rho_1 \le \rho_2, \rho_2 \preceq^i_+ \rho_3\} & \cup \Rightarrow & \{\rho_0 \le \rho_3\} \\
C \cup \{\ell_1 \preceq^i_- \ell_0, \ell_1 \le \ell_2, \ell_2 \preceq^i_+ \ell_3\} & \cup \Rightarrow & \{\ell_0 \le \ell_3\}
\end{array}
$$

Fig. 28. Extensions to Figs. 8 and 21 for context sensitivity

corresponds to an *output* from a function, and we draw it with an edge $\eta \to^{)i} \eta'$. Similarly, a constraint $\eta \preceq^i_- \eta'$ corresponds to an *input* to a function, and we draw it with an edge $\eta' \to^{(i} \eta'$. Notice that for a negative polarity constraint, the direction of the graph edge is opposite the direction of the $\preceq$.

Fig. 27(b) shows the new type inference rules.[8] Let first types v_1, and then types e_2 with f bound to the type scheme $(\forall.\tau_1, \vec{\eta})$, where τ_1 is the type of v_1, and $\vec{\eta}$ is the set of free labels of Γ (as usual for Hindley-Milner-style polymorphism, these are the labels we cannot quantify [Pierce 2002]). In Inst, we instantiate a type scheme $(\forall.\tau, \vec{\eta})$ at index i. We generate a type τ' by reannotating τ with fresh labels. We then generate an instantiation constraint $\tau \preceq^i_+ \tau'$ to indicate that τ is used at index i at type τ', and we generate constraints $\vec{\eta} \preceq^i_{\pm} \vec{\eta}$ to indicate that the substitution S_i represented by the constraint $\tau \preceq^i_+ \tau'$ must not rename any variables in $\vec{\eta}$, i.e., they must be instantiated to themselves. (Here the $\pm$ is shorthand for generating two constraints, one with polarity $+$ and one with polarity $-$.) Lastly, Fix combines Let and Inst, binding f to a type scheme during the typing of v, and then instantiating f to a fresh type as the result.

6.3 Context-Sensitive Label Flow Constraint Resolution

To compute the flow of labels in our new constraint system, we extend our constraint rewriting rules as shown in Fig. 28. The first set of rewrite rules corresponds to the standard subtyping rules. We reduce $\preceq^i_p$ constraints to components of a type in a manner that is invariant for references and pairs (due to lazy fields, and thus analogously to equating pair labels in Fig. 21), and co- and contra-variant for function return and argument types, respectively. Here we write $\bar{p}$ for the opposite of polarity p.

The next two rules propagate instantiation constraints to components of a lazy pair. The first rule requires that if ζ is instantiated to ζ', then the j component of ζ is instantiated to the j component of ζ'. The next rule handles the other direction of instantiation. Note that in both rules, the generated constraint on the right-hand

[8] We have implicitly relaxed the definition of Γ to also include type schemes σ.

side refers to $\zeta'[j]$ although that might be empty. In that case we assume that it is set to $\langle\langle\tau\rangle\rangle$ (not shown), i.e., a type of the appropriate shape, annotated with fresh labels.

The last two rewrite rules propagate constraints along paths with matched parentheses. Pictorially, given a sequence of constraints $\rho_0 \rightarrow^{(i} \rho_1 \rightarrow \rho_2 \rightarrow^{)i} \rho_3$, the first rewrite rule generates a new constraint $\rho_0 \rightarrow \rho_3$, derived from matching the parentheses on the path; this is called *matched flow* by Rehof and Fähndrich [2001]. The last rewriting rule follows the same pattern for abstract locks.

Given these rewriting rules, our definition of $Flow()$ remains the same:

$$\begin{aligned} Flow(C,\rho) &= \{\rho' \mid \rho' \leq \rho \in Sol(C)\} \\ Flow(C,\ell) &= \{\ell' \mid \ell' \leq \ell \in Sol(C)\} \end{aligned}$$

For example, letting C be the constraints in Fig. 25, we have $Flow(C,\rho x) = \{\rho 1\}$ and $Flow(C,\rho y) = \{\rho 2\}$.

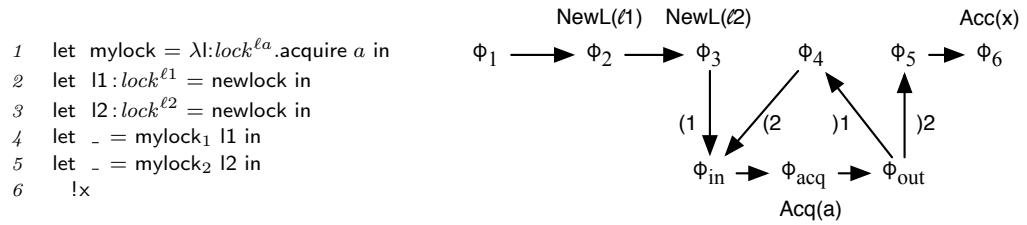
6.4 Context-sensitive Data Flow Analysis

We show now how we extend the dataflow analysis and ACFG with context sensitivity, encoding it also with parametric polymorphism.

As discussed above, each instantiation of a type scheme $\sigma = (\forall.\tau, \vec{\eta})$ at index i generates the constraint $\tau \preceq^i_+ \tau'$, where $\tau' = \langle\langle\tau\rangle\rangle$. We say that τ is the *abstract* type and τ' is the *instance* type at the instantiation i. Moreover, the instantiation i defines a substitution S_i of labels, such that $S_i(\tau) = \tau'$ and also for all labels $\eta \in \vec{\eta}$, we have $S_i(\eta) = \eta$. We represent the reverse substitution with S_i^{-1}, mapping the labels of τ' to τ. For example, the instantiation $ref^{\,\rho}(int) \preceq^i_+ ref^{\,\rho'}(int)$ defines the substitutions $S_i(\rho) = \rho'$ and $S_i^{-1}(\rho') = \rho$. Note that the substitutions S_i and S_i^{-1} only translate between the abstract and the instance type, regardless of the instantiation polarity. So, even if the above instantiation had negative polarity, $ref^{\,\rho}(int) \preceq^i_- ref^{\,\rho'}(int)$, the substitutions remain $S_i(\rho) = \rho'$ and $S_i^{-1}(\rho') = \rho$.

Consider an instantiation of a function type according to Fig. 28, $(\tau_1, \phi_1) \rightarrow (\tau_1', \phi_1') \preceq^i_+ (\tau_2, \phi_2) \rightarrow (\tau_2', \phi_2')$. This generates the constraints $\phi_1 \preceq^i_- \phi_2$ and $\phi_1' \preceq^i_+ \phi_2'$ among the statement labels representing the function start and end of the abstract and instance types. We extend the definition of dataflow analysis on the ACFG to account for the two kinds of instantiation edges, so that when we propagate facts across instantiation edges, they are brought to the correct context. Namely, we apply the substitution S_i to all facts propagated from ϕ_1' to ϕ_2', to translate all labels defined in the context of ϕ_1' to the corresponding labels in the context of ϕ_2'. Conversely, we apply the substitution S_i^{-1} to all facts propagated from ϕ_2 to ϕ_1 (recall that the negative instantiation polarity reverses the direction of the graph edge), to translate all facts in terms of the abstract type of the instantiation. Note that S_i is a partial map, so it is possible that not all facts defined at the left side of the instantiation can be expressed in terms of its right side, or vice versa. In general, we only propagate the facts that can be expressed at the target statement label of an instantiation edge.

Although these rules might resemble the Call and Ret kinds and edges in the control flow graph, in fact they are orthogonal. Specifically, an instantiation edge between statement labels corresponds to an occurrence of a function name in the program, whereas statement labels with kind Call and Ret correspond to a function



(a) Source program (b) ACFG

Fig. 29. Context-sensitive analysis for lock state

invocation. In many cases these happen to coincide, but one does not imply the other in general. For instance, when a program uses a function pointer to alias many functions and invokes it once, then many instantiations correspond to one invocation, whereas when the program assigns one function to a function pointer, but invokes it many times, then one instantiation has many invocations.

Lock State. In the Lock State Analysis we propagate the set of held locks across instantiation edges as discussed above, by applying the appropriate renaming, according to the polarity of the constraint. Namely, for an instantiation $\phi \preceq^i_+ \phi'$ that corresponds to $\phi \rightarrow^{)i} \phi'$, we translate the set of held locks at ϕ by applying the substitution S_i, we close the translated set under aliasing, and we propagate the resulting set of held locks (and all their aliases) to statement ϕ:

$$Flow(C, S_i(Acq_{out}(\phi))) \subseteq Acq_{in}(\phi')$$

Similarly, for an instantiation $\phi \preceq^i_- \phi'$ that corresponds to $\phi' \rightarrow^{(i} \phi$, we translate the set of held locks at ϕ' by applying the substitution S_i^{-1} before propagating it to ϕ:

$$Flow(C, S_i^{-1}(Acq_{out}(\phi'))) \subseteq Acq_{in}(\phi)$$

For example, the program in Fig. 29(a) defines a wrapper function for acquiring a lock that takes an argument a of type $lock^{\ell a}$ and acquires it. The program creates two locks and acquires them before dereferencing a variable x (not defined here, for brevity). Clearly, since the function mylock acquires its argument, the mechanism for "hiding" irrelevant locks using Call and Ret nodes has no effect here. Indeed, we need to differentiate between the two contexts of the calls to mylock (marked with indices 1 and 2) to infer that both locks l1 and l2 are held at the dereference point. We do this using the context-sensitive ACFG shown in Fig. 29(b), simplified by omitting nodes of no interest for this example. During the dataflow analysis, we infer (as in the monomorphic case) that at the end of the function (ϕ_{out}) the abstract lock ℓ is held ($\ell \in Acq_{out}(\phi_{out})$). We also have $\phi_{out} \preceq^1_+ \phi_4$ and $\phi_{out} \preceq^2_+ \phi_5$. Moreover, from the instantiations 1 and 2 of mylock's type $(lock^{\ell a}, \phi_{in}) \rightarrow (int, \phi_{out})$, we have $S_1(\ell a) = \ell 1$ and $S_2(\ell a) = \ell 2$. To propagate the set of held locks along the instantiation edges $\phi_{out} \preceq^i_+ \phi_4$, we apply the corresponding substitution to the set of held locks, propagating $S_1(Acq_{out}(\phi_{out}) = S_1(\{\ell a\}) = \{\ell 1\}$ to ϕ_4. Similarly, we propagate $S_2(Acq_{out}(\phi_{out}) = S_2(\{\ell a\}) = \{\ell 2\}$ to ϕ_5.

$$\textsc{Eff-Lam}\quad \frac{\Phi_f; \Gamma, x : \tau' \vdash e : \tau}{\Phi; \Gamma \vdash \lambda x{:}e. : \tau' \to^{\Phi_f} \tau}$$

(a) Type rule for function definition

$$C \cup \{\tau_1 \to^{\Phi_1} \tau_1' \preceq_p^i \tau_2 \to^{\Phi_2} \tau_2'\} \Rightarrow C \cup \{\tau_1 \preceq_{\bar{p}}^i \tau_2, \tau_1' \preceq_p^i \tau_2', \Phi_1 \preceq_p^i \Phi_2\}$$

$$C \cup \{\Phi_1 \preceq_p^i \Phi_2\} \Rightarrow C \cup \{\Phi^\alpha \preceq_p^i \Phi^\alpha, \Phi^\omega \preceq_{\bar{p}}^i \Phi^\omega\}$$

(b) Constraint resolution rules

Fig. 30. Context-Sensitive Contextual Effects

Correlation Inference. We extend the correlation inference with context sensitivity in a similar way, adapted for a backwards analysis. Specifically, at instantiation $\phi \preceq_+^i \phi'$, due to the backwards direction of the propagation, we propagate from ϕ' to ϕ. Since ϕ' lies in the "instance" context, we use S_i^{-1} to translate the state at ϕ' to the state at ϕ. Namely, for every correlation $\rho \rhd \vec{\ell}$ at ϕ', we add a correlation $S_i^{-1}(\rho) \rhd \mathit{Flow}(C, S_i^{-1}(\vec{\ell}))$ to ϕ.

Likewise, for negative instantiation edges $\phi \preceq_-^i \phi'$, we propagate from ϕ to ϕ'. As in this case ϕ lies in the left side of the instantiation, we use S_i to translate the state at ϕ to the state at ϕ'. Now, for every correlation $\rho \rhd \vec{\ell}$ at ϕ, we add a correlation $S_i(\rho) \rhd \mathit{Flow}(C, S_i(\vec{\ell}))$ to ϕ.

6.5 Context-sensitive Sharing Analysis

We extended the sharing analysis with context sensitivity, both for computing the shared locations at fork points using context-sensitive contextual effects, and also for the flow-sensitive propagation of sharing information that marks the interesting dereferences in the program.

Contextual Effects. The contextual effect system presented in Section 4.1 can be extended with context sensitivity in the same way as the label flow analysis. As presented in detail in previous work [Neamtiu et al. 2008], function types are annotated with the effect Φ_f of the function. We repeat the type rule for function definition in Fig. 30(a). Note that a function type is annotated with the effect Φ_f of the function body. Moreover, since function definition itself has no effect, it can be typed under any effect Φ. As in Section 4.1, we present contextual effects as a standalone system, although it is straightforward to combine with the rules in Fig. 27. Fig. 30(b) defines the instantiation for annotated function types and contextual effects. The contextual effect of a function is instantiated covariantly, which translates to a covariant instantiation for the standard effect, and a contravariant instantiation for the future effect, because the standard effects of the function are defined inside the function and "returned" to the environment, whereas the future effect is defined outside the function, in the calling contexts, and "enters" the function.

Note that the future effect ω at a given program point in a function includes

the effects of the program after the function returns. In combination with context sensitivity this might cause some locations that are in the effect (i.e. accessed after the current function returns) to not have a corresponding location, or not yet exist, in the current context. In other words, there might not be a matched parenthesis path from a location ρ, dereferenced in the continuation, to the future effect ω in the current location. For example, consider the toy program:

```
1  let f = λ x . x+1 in
2    f1 1;
3  let p = (ref 41) in
4    !p
```

Clearly, the variable p is not in scope in the body of function f, and moreover, there is no alias of p that is in scope either. This means that there can be no matched parentheses path from p to the future effect of expression $\mathsf{x}+1$ in the body of f. Indeed, the only path from p to the future effect of the expression involves a (*1* edge due to the instantiation of f. However, p is clearly in the future effect of the expression $\mathsf{x}+1$, as it is dereferenced later in the program, after the call to f. To address this problem, when solving for future effects at fork points to compute shared locations, we consider paths that do not contain mismatched parentheses (a.k.a. PN-flow [Fähndrich et al. 2000]), instead of paths with only matched parentheses. For the same reason, we also use PN-flow to compute the set of labels in scope and their aliases for the scoping optimization discussed in Section 4.2.

Shared Locations Propagation. The propagation of shared locations according to dataflow discussed in Section 4.3 is straightforward to extend with context sensitivity, in the same way as the above dataflow analyses for lock state and correlation inference. For positive instantiation edges, $\phi \preceq^i_+ \phi'$, we propagate from ϕ to ϕ' (forwards analysis), using S_i to translate the set of shared locations at ϕ to the context of ϕ' and adding the closed set (to account for aliasing) to the state at ϕ':

$$\mathit{Flow}(C, S_i(\mathit{Sh}_{out}(\phi))) \subseteq \mathit{Sh}_{out}(\phi')$$

Similarly, for negative instantiation edges $\phi \preceq^i_- \phi'$, we propagate from ϕ' to ϕ using S_i to translate the shared locations to the context of ϕ:

$$\mathit{Flow}(C, S_i^{-1}(\mathit{Sh}_{out}(\phi'))) \subseteq \mathit{Sh}_{out}(\phi)$$

6.6 Results

Fig. 31 compares the running times and number of warnings for context-sensitive and context-insensitive versions of LOCKSMITH. Note that since the context-sensitive analysis is no less sound than the context-insensitive analysis, any warning it eliminates is a false positive. The context-sensitive results are the same as Fig. 4, reproduced here for convenience. These results show that context sensitivity significantly increases the running time of the analysis, often very significantly, e.g., for most of the Linux drivers. The exceptions are the sis900 and slip benchmarks, for which the imprecision of context-insensitive analysis creates so much aliasing that LOCKSMITH runs out of memory trying to compute the closure of the label flow

	Context-sensitive		Context-insensitive	
Benchmark	Time (s)	Warnings	Time (s)	Warnings
aget	0.85	62	0.64	77
ctrace	0.59	10	0.42	21
engine	0.88	7	0.60	15
knot	0.78	12	0.60	31
pfscan	0.46	6	0.50	26
smtprc	5.37	46	4.16	128
3c501	9.18	15	0.75	20
eql	21.38	35	0.86	41
hp100	143.23	14	2.76	25
plip	19.14	42	1.39	46
sis900	71.03	6	Out of Mem.	n/a
slip	16.99	3	Out of Mem.	n/a
sundance	106.79	5	1.32	20
synclink	1521.07	139	23.42	227
wavelan	19.70	10	9.59	143

Fig. 31. Comparison of context sensitivity and insensitivity

graph. Furthermore, we see that context sensitivity notably reduces the number of warnings reported by LOCKSMITH, eliminating many false positives.

7. RELATED WORK

Several systems have been developed for detecting data races and other concurrency errors in multi-threaded programs, including dynamic analysis, static analysis, and hybrid systems.

Dynamic systems such as Eraser [Savage et al. 1997] instrument a program to find data races at run time and require no annotations. The efficiency and precision of dynamic systems can be improved with static analysis [Choi et al. 2002; O'Callahan and Choi 2003; Agarwal et al. 2005]. Dynamic systems are fast and easy to use, but cannot prove the absence of races, and require comprehensive test suites.

Researchers have developed type checking systems against races [Flanagan and Abadi 1999] for several languages, including Java [Flanagan and Freund 2000], Java variants [Boyapati and Rinard 2001], and Cyclone [Grossman 2003]. Such systems based on type checking perform very well but require a significant number of programmer annotations, which can be time consuming when checking large code bases [Engler and Ashcraft 2003; Flanagan and Freund 2001]. Static race detection in ESC/Java [Flanagan et al. 2002], which employs a theorem prover, similarly requires many annotations.

Some researchers have developed tools to automatically infer the annotations needed by the Java-based type checking systems just mentioned. Most target Java 1.4, which simplifies the problem by permitting only lexically-acquired locks via synchronized statements, whereas C (and Java 1.5) programs may acquire and release locks at any program point. Houdini [Flanagan and Freund 2001] can infer types for the original race-free Java system [Flanagan and Freund 2000], but lacks context sensitivity. More recently Agarwal and Stoller [Agarwal and Stoller 2004] and Rose et al [Rose et al. 2005] have developed algorithms that infer types based on dynamic traces, but these require sizeable test suites to avoid excessive false

alarms. Flanagan and Freund [Flanagan and Freund 2007] have proposed a system for inference which is formulated to support parameterized classes and dependent types. Though the problem is NP-complete, their SAT-based approach can analyze 30K lines of Java code in 46 minutes. Von Praun and Gross's dataflow-based system [von Praun and Gross 2003] also requires no annotations and performs well, checking 2000-line programs in a few seconds.

Naik, Aiken, and Whaley present a race detection system for Java [Naik et al. 2006]. Their system scales well to large Java programs and has found several races. Analyzing Java 1.4 avoids some problems we encountered analyzing C code, such as flow-sensitive locking, low-level pointer operations, and unsafe type casts. They also omit linearity checking, which we include in LOCKSMITH. In later work [Naik and Aiken 2007], Naik and Aiken address the lack of linearity in locks by introducing a *conditional must not alias* analysis, which can handle fine-grain locking.

Several completely automatic static analyses have been developed for finding races in C code. Polyspace [Hote 2004] is a proprietary tool that uses abstract interpretation to find data races (and other problems). The BLAST model checker has been used to find data races in programs written in NesC, a variant of C [Henzinger et al. 2004]. Race checking is not limited to checking for consistent correlation and can be state dependent, but is limited to checking global variables and can be quite expensive. Seidl et al [Seidl et al. 2003] propose a framework for analyzing multithreaded programs that interact through global variables. Using their framework they develop a race detection system for C and apply it to a small set of benchmarks, finding several data races. It is unclear whether their analysis supports context sensitivity and how it models data structures. RacerX [Engler and Ashcraft 2003] does not soundly model some features of C for better scalability and to reduce false alarms, but may miss races as a result. KISS [Qadeer and Wu 2004] builds on model checking techniques, and has been shown to find many races, but ignores certain kinds of thread interleavings.

Voung, Jhala and Lerner present RELAY, a race detection system for C [Voung et al. 2007] that uses flow-sensitive propagation of lock set and guarded-by information similar to LOCKSMITH. RELAY scales to millions of lines of C code by analyzing and summarizing the behavior of parts of the program in parallel, using symbolic evaluation. Unlike RELAY, LOCKSMITH generates and solves the constraints for the whole program together, and is implemented to run on a single processor, limiting its scalability. One benefit of the whole program, type-based analysis in LOCKSMITH is that it can track the flow of function pointers precisely. In contrast, the modular per-file analysis in RELAY may not track aliasing of function pointers across files correctly, which can result in unmodeled control flow.

Terauchi proposes LP-Race [Terauchi 2008], a static analysis tool that reduces the problem of race detection to linear programming. The reduction is such that one need not directly compute acquired locks, and LP-race can handle synchronization via semaphores and signals. LP-Race scales to medium-sized programs, some of which cause LOCKSMITH to run out of memory. However, LOCKSMITH runs slightly faster though it uses a more precise aliasing and sharing analysis. We conjecture that, as a result of this more precise analysis, LOCKSMITH's reports are more precise—two abstract locations differentiated by a precise analysis could

be considered to be one location in a less precise analysis. LOCKSMITH's analysis is inclusion-based, and is both field- and context-sensitive. LP-Race uses a unification-based analysis, that is also field-sensitive. In one common benchmark (smtprc) LP-Race was able to eliminate false positives due to handling semaphores and thread joins. However, LP-Race produced additional false positives due to a limitation in handling loops that fork an unbounded number of threads. Due to the way we use future effects in our sharing analysis, LOCKSMITH is able to handle such loops and infer shared locations more precisely.

Work that detects violations of *atomicity*, either dynamically [Flanagan and Freund 2004] or statically [Flanagan and Qadeer 2003; Flanagan et al. 2005] typically requires a program to be free of races.

Our analysis is based on ideas initially explored by Reps et al [Reps et al. 1995] and Rehof and Fähndrich [Rehof and Fähndrich 2001], who showed how to encode context-sensitive analysis as a context-free language reachability problem. Our support for existential types is related to `restrict` or `focus` for alias analysis [Aiken et al. 2003; Fahndrich and DeLine 2002]. Our flow-sensitive analysis is a significant extension of our previous work on flow-sensitive type qualifiers [Foster et al. 2002], which used a similar flow-sensitive constraint graph. Both systems can be seen as inference for a variant of the calculus of capabilities [Crary et al. 1999].

Correlation between locks and locations is similar to correlation between regions and pointers, and several researchers have looked at the problem of region inference, including the Tofte and Birkedal system for the ML Kit [Tofte and Birkedal 1998]. Henglein et al [Henglein et al. 2001] use a control-flow-sensitive and context-sensitive type system to check that regions with non-lexical allocation and deallocation are used correctly. Our treatment of lock allocation is similar to Henglein et al's treatment of region allocation, but our formal system supports higher-order functions, and we present a constraint-based inference algorithm.

8. CONCLUSION

In this paper we described LOCKSMITH, a static analysis tool for finding data races in C programs. We presented the core algorithms of LOCKSMITH on a simple imperative language and explored the engineering challenges in handling all of C. For each algorithm we performed extensive measurements, comparing with alternative algorithms in terms of precision and performance. We found that context sensitivity greatly reduces the number of false warnings, but also limits LOCKSMITH's overall scalability. Perhaps surprisingly, we found that field sensitivity improves both precision and performance, the latter because more precise modeling of aliasing speeds up subsequent phases of LOCKSMITH. We described our approaches to modeling fields lazily and for handling `void` $*$ pointers, both of which were important to precision and performance. We also found that our sharing analysis was effective, determining that most locations are thread-local, and that some simple scoping optimizations and a uniqueness analysis improved on our sharing analysis further. Lastly, we found that not using a worklist was the best strategy for our dataflow analyses. We believe these results will prove valuable to designers of other static analyses for C.

ACKNOWLEDGMENTS

We would like to thank Alan Cox for answering a question on Linux device drivers. This work was supported by NSF CCF-0541036, CCF-0346989, CCF-0430118, and CCF-0524036.

REFERENCES

Agarwal, R., Sasturkar, A., Wang, L., and Stoller, S. D. 2005. Optimized run-time race detection and atomicity checking using partial discovered types. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM Press, New York, NY, USA, 233–242.

Agarwal, R. and Stoller, S. D. 2004. Type inference for parameterized race-free java. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*. Lecture Notes in Computer Science, vol. 2937. Springer-Verlag, Venice, Italy, 149–160.

Aho, A. V. and Ullman, J. D. 1977. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Aiken, A., Foster, J. S., Kodumal, J., and Terauchi, T. 2003. Checking and inferring local non-aliasing. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 129–140.

Alexandrescu, A., Boehm, H., Henney, K., Hutchings, B., Lea, D., and Pugh, B. 2005. Memory model for multithreaded c++: Issues.

Boyapati, C. and Rinard, M. 2001. A parameterized type system for race-free java programs. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*. ACM Press, New York, NY, USA, 56–69.

Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 258–269.

Cooper, K. D., Harvey, T. J., and Kennedy, K. 2004. Iterative data-flow analysis, revisited. Tech. Rep. TR04-100, Department of Computer Science, Rice University.

Crary, K., Walker, D., and Morrisett, G. 1999. Typed memory management in a calculus of capabilities. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 262–275.

Engler, D. and Ashcraft, K. 2003. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM Press, New York, NY, USA, 237–252.

Fahndrich, M. and DeLine, R. 2002. Adoption and focus: practical linear types for imperative programming. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 13–24.

Fähndrich, M., Rehof, J., and Das, M. 2000. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. ACM, New York, NY, USA, 253–263.

Flanagan, C. and Abadi, M. 1999. Types for safe locking. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*. Springer-Verlag, London, UK, 91–108.

Flanagan, C. and Freund, S. N. 2000. Type-based race detection for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 219–232.

Flanagan, C. and Freund, S. N. 2001. Detecting race conditions in large programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM Press, New York, NY, USA, 90–96.

Flanagan, C. and Freund, S. N. 2004. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 256–267.

Flanagan, C. and Freund, S. N. 2007. Type inference against races. *Sci. Comput. Program. 64,* 1, 140–165.

Flanagan, C., Freund, S. N., and Lifshin, M. 2005. Type inference for atomicity. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*. ACM Press, New York, NY, USA, 47–58.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. 2002. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 234–245.

Flanagan, C. and Qadeer, S. 2003. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 338–349.

Foster, J. S., Johnson, R., Kodumal, J., and Aiken, A. 2006. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst. 28,* 6, 1035–1087.

Foster, J. S., Terauchi, T., and Aiken, A. 2002. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 1–12.

Grossman, D. 2003. Type-safe multithreading in cyclone. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*. ACM Press, New York, NY, USA, 13–25.

Heintze, N. and Tardieu, O. 2001. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. ACM, New York, NY, USA, 254–263.

Henglein, F. 1993. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst. 15,* 2, 253–289.

Henglein, F., Makholm, H., and Niss, H. 2001. A direct approach to control-flow sensitive region-based memory management. In *PPDP '01: Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM Press, New York, NY, USA, 175–186.

Henzinger, T. A., Jhala, R., and Majumdar, R. 2004. Race checking by context inference. *SIGPLAN Not. 39,* 6, 1–13.

Hote, C. 2004. Run-time error detection through semantic analysis.

intel.com. 2007. Teraflops research chip.

Johnson, R. and Wagner, D. 2004. Finding user/kernel pointer bugs with type inference. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 9–9.

Kodumal, J. and Aiken, A. 2004. The set constraint/cfl reachability connection in practice. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. ACM, New York, NY, USA, 207–218.

Kodumal, J. and Aiken, A. 2005. Banshee: A scalable constraint-based analysis toolkit. In *SAS*, C. Hankin and I. Siveroni, Eds. Lecture Notes in Computer Science, vol. 3672. Springer, London, UK, 218–234.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7, 558–565.

Leveson, N. G. and Turner, C. S. 1993. An investigation of the therac-25 accidents. *Computer 26,* 7, 18–41.

Manson, J., Pugh, W., and Adve, S. V. 2005. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 378–391.

Mossin, C. 1996. Flow Analysis of Typed Higher-Order Programs. Ph.D. thesis, DIKU, Department of Computer Science, University of Copenhagen.

Naik, M. and Aiken, A. 2007. Conditional must not aliasing for static race detection. *SIGPLAN Not. 42,* 1, 327–338.

Naik, M., Aiken, A., and Whaley, J. 2006. Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 308–319.

Neamtiu, I., Hicks, M., Foster, J. S., and Pratikakis, P. 2008. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 37–50.

Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. 2002. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*. Springer-Verlag, London, UK, 213–228.

news.com. 2007. Designer puts 96 cores on single chip.

O'Callahan, R. and Choi, J.-D. 2003. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, New York, NY, USA, 167–178.

Pierce, B. C. 2002. *Types and programming languages*. MIT Press, Cambridge, MA, USA.

Poulsen, K. 2004. Tracking the blackout bug.

Pratikakis, P., Foster, J. S., and Hicks, M. 2006a. Existential label flow inference via CFL reachability. In *Proceedings of the Static Analysis Symposium (SAS)*. Springer, Seoul, Korea, 88–106.

Pratikakis, P., Foster, J. S., and Hicks, M. 2006b. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 320–331.

Qadeer, S. and Wu, D. 2004. Kiss: keep it simple and sequential. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 14–24.

Rehof, J. and Fähndrich, M. 2001. Type-base flow analysis: from polymorphic subtyping to cfl-reachability. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 54–66.

Reps, T., Horwitz, S., and Sagiv, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 49–61.

Reynolds, J. C. 2004. Toward a grainless semantics for shared-variable concurrency. In *FSTTCS*, K. Lodaya and M. Mahajan, Eds. Lecture Notes in Computer Science, vol. 3328. Springer, Chennai, India, 35–48.

Rose, J., Swamy, N., and Hicks, M. 2005. Dynamic inference of polymorphic lock types. *Science of Computer Programming (SCP) 58,* 3 (December), 366–383. Special Issue on Concurrency and Synchronization in Java programs. Supercedes 2004 CSJP paper of the same name.

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15,* 4, 391–411.

Seidl, H., Vene, V., and Müller-Olm, M. 2003. Global invariants for analyzing multi-threaded applications.

Siff, M., Chandra, S., Ball, T., Kunchithapadam, K., and Reps, T. 1999. Coping with type casts in c. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*. Springer-Verlag, London, UK, 180–198.

Smith, F., Walker, D., and Morrisett, J. G. 2000. Alias types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*. Springer-Verlag, London, UK, 366–381.

Talpin, J.-P. and Jouvelot, P. 1994. The type and effect discipline. *Inf. Comput. 111,* 2, 245–296.

Terauchi, T. 2008. Checking race freedom via linear programming. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 1–10.

Tofte, M. and Birkedal, L. 1998. A region inference algorithm. *ACM Trans. Program. Lang. Syst. 20,* 4, 724–767.

von Praun, C. and Gross, T. R. 2003. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM, New York, NY, USA, 115–128.

Voung, J. W., Jhala, R., and Lerner, S. 2007. Relay: static race detection on millions of lines of code. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, New York, NY, USA, 205–214.