# Static Race Detection for Device Drivers: The Goblint Approach

Vesal Vojdani
University of Tartu, Estonia

Kalmer Apinis
University of Tartu, Estonia

Vootele Rõtov
University of Tartu, Estonia

Helmut Seidl
Technische Universität München, Germany

Varmo Vene
University of Tartu, Estonia

Ralf Vogler
Technische Universität München, Germany

## ABSTRACT

Device drivers rely on fine-grained locking to ensure safe access to shared data structures. For human testers, concurrency makes such code notoriously hard to debug; for automated reasoning, dynamically allocated memory and low-level pointer manipulation poses significant challenges. We present a flexible approach to data race analysis, implemented in the open source Goblint static analysis framework that combines different pointer and value analyses in order to handle a wide range of locking idioms, including locks allocated dynamically as well as locks stored in arrays. To the best of our knowledge, this is the most ambitious effort, having lasted well over ten years, to create a fully automated static race detection tool that can deal with most of the intricate locking schemes found in Linux device drivers. Our evaluation shows that these analyses are sufficiently precise, but practical use of these techniques requires inferring environmental and domain-specific assumptions.

## CCS Concepts

**•Theory of computation → Program analysis; •Software and its engineering → Software safety;**

## Keywords

Concurrency, race condition, abstract interpretation

## 1. INTRODUCTION

A *multiple access data race* occurs in a concurrent program when different threads simultaneously attempt to access a shared memory location and one of the accesses is a write operation. Without proper synchronization, the result of such accesses are unpredictable. As a problem notoriously hard to debug, there has been plenty of effort to detect such bugs statically [23, 24, 34, 46]. However, fully automated static analyzers have had limited success in challenging settings, such as analyzing device drivers.

The basic approach to race detection is to track the set of locks definitely held by each thread, and so ensure that all accesses to shared memory locations are protected by a common lock. This so-called *lockset algorithm* was first applied in a dynamic setting [14]. There are, however, serious difficulties when adapting this idea to the static analysis of Linux device drivers. In low-level C code, locks can be placed within heap-allocated data structures, as depicted in Figure 1. Elements within a linked list have their own locks protecting their data fields; however, the linked list itself is protected by a list lock. As the Linux kernel lists have their fields embedded into the container struct, we have different portions of the same structure being protected with different locks of different granularity. Since precise shape-analysis is costly for the low-level linked lists provided by the Linux list API [25, 26], we use a combination of aliasing and points-to analyses each targeting locking schemes of different granularities.

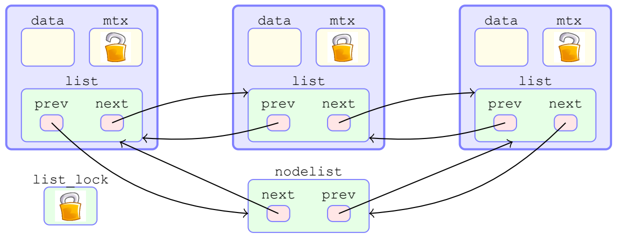

**Figure 1: Mixed-granularity locking scheme.**

Linux code also features complicated control flow. Conditional locking, possibly failing locks, and value-dependent synchronization have been reported as major sources of false alarms for race detectors [37, 46]. These constructs require that the values of program variables be taken into account in a concurrent setting. Further, when locks are stored in arrays, we need relational analyses to correlate lock indices with the data they protect.

We present an algorithm for static race detection that can deal with value-dependent synchronization, and we then extend this to deal with several locking schemes for dynamically allocated data. Most crucially, the approach we present in this paper is parametric on the precise analyses used. Race freedom is a conditional proposition that requires the combination of may-alias information with must-equality information. We show how such information can be combined in a modular and generic fashion. This extensibility has allowed us to keep improving the framework, adapting it to different concurrency models, as well as considering increasingly complicated locking idioms.

The algorithm described in the paper was implemented in the Goblint tool, which was used to evaluate the approach. The Goblint

*ASE'16*, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970337

is an exhaustive bug detection tool: it relies on sound data flow analyses using abstract domains to conservatively over-approximate the behavior of the system. Thus, when the analyzer is satisfied, we have high confidence in the correctness of the analyzed code, but the natural drawback is that we may have many spurious warnings. To get a realistic impression of its ability, we chose as our analysis targets all Linux character device drivers which our front-end library could digest. The results show that on average less than 5% of memory locations are ruled potential races where the access is directly in the source code of the driver. While we were not able to exclude these cases as potential races upon manual inspection, we did not find any races in currently maintained device drivers that we were convinced could lead to a crash or corruption of data. For a practically useful tool, we still need to reduce the number of false alarms by taking domain-specific and environmental assumptions into account. In summary, this paper makes the following contributions:

- We present a sound concurrency abstraction, based on *privatization*, that can detect races and compute the value of integer variables (Section 3).
- We extend the privatization approach from static integers variables to generic heap regions. We present a race detection algorithm parametric on may- and must-alias analyses that can deal with a wide range of locking schemes (Sections 4 and 5).
- We have implemented this approach in the open-source Goblint analyzer[1] and evaluated this approach on a set of device drivers (Section 6).

We will begin by presenting the consistency model that architecture-independent code in the Linux kernel may assume. This will make our analysis much clearer because there is a correspondence between what our analysis computes and the configurations of the model. Section 4 illustrates our approach with many examples.

## 2. CONSISTENCY MODEL

The Linux operating system supports many different architectures, each with differing models of memory consistency. Outside architecture-specific code, we can make few assumptions about when read and write operations are visible to different threads. In this section, we describe a model of concurrent execution that only makes the consistency assumptions satisfied by all architectures.

As we analyze Linux kernel modules separately, we model each driver as an open program operating in a hostile environment. Each module contains an initialization function where execution of the module begins. This function will eventually register a set of callback functions and interrupt handlers with the environment which can then call these exported functions at will. From the moment these functions are registered, we assume they can potentially run in parallel. In reality, there are more refined scheduling constraints; e.g., probe functions and open/close run sequentially for each device. Although our concurrency model over-approximates the data flow, we can still consider more refined may-run-in-parallel information when reporting data races, as will be seen in Example 5.

In our model, whenever the kernel calls the operations of the device driver, the associated callback function runs in a new thread instance. Following the terminology of Deligiannis et al. [13], the registered functions form the set of thread *templates* $\mathcal{T}$. The code of each thread template $t \in \mathcal{T}$ is given as a control flow graph $G_t = (N, E, n_t)$ where $N$ is the set of program points, $n_t$ the start point of this thread template, and each edge $(u, l, v) \in E$ is a transition labeled with either an elementary command $s$ or one of the primitive concurrency operations for acquiring and releasing mutexes:

$$l ::= s \mid \mathsf{lock}(m) \mid \mathsf{unlock}(m)\,.$$

Since this paper focuses exclusively on lock-based synchronization, we do not include in our formalism any constructs for threads to wait on each other without the use of mutexes, such as wait queues, events and conditional variables.

As an unbounded number of instances of each thread template may run in parallel, the set of thread identifiers is $\mathcal{I} = \mathcal{T} \times \mathbb{N}$. Henceforth, when we use the word "thread", we specifically mean these thread instances identified by the elements in $\mathcal{I}$. Thus, in our formal model, we assume that after execution of some initialization code, we have an infinite set of threads $\mathcal{I}$ ready to run in parallel. These threads issue instructions that involve memory accesses, including the synchronization operations. In a multi-processor system, such instructions are not completed instantly. The following recounts the standard terminology.

*Definition 1.* An instruction is *completed with respect to another thread* when that thread can read the new value, in the case of a store, or no longer influence the result, in the case of a read. An instruction is *completed* when it is completed with respect to all other threads. There has to be some memory consistency between the views of each thread; this is determined by the *consistency model* of the architecture.

As mentioned above, we have to assume the weakest consistency model common to all supported architectures. The Linux memory model only provides the guarantees of *release consistency* [15]. Under this consistency model, synchronization operations are distinguished as acquire and release operations, and they only serve as one-way barriers. For example, memory operations issued after an acquire will be completed after the acquire operation has completed; however, memory operations issued before the acquire may be completed after the acquire operation has completed. We describe the model at a high level, leaving the details of the cache coherence protocol and its implementation to indeterminism. We only enforce that, at worst, caches need to be flushed upon release and invalidated upon acquire.

Let $\mathcal{L}$ denote the address space, i.e., the set of memory locations, available to the program. To reduce notational overhead, we treat local variables as thread-local globals in this formalization. The state of a single-threaded program could be described by its program counter, indicating the location $u \in N$ of the next command to be executed, and a mapping from memory locations to their values. As the intra-thread semantics is of little interest to us, we shall just view memory locations as holding integer values. The state of memory is thus represented as a mapping in $\mathcal{D} = \mathcal{L} \to \mathbb{N}$.

The state of executing a multi-threaded program is represented in our model by the local states of each of the threads in $\mathcal{I}$ as well as shared state $\varphi \in \mathcal{D}$, representing, e.g., physical memory or the value at the home node, depending on the architecture. In our model, commands are issued in their intra-thread order, although the order in which they complete is non-deterministic. The state of an individual thread $i \in \mathcal{I}$ is thus characterized by its program counter $u_i$, indicating the command it is about to issue, and the thread's local (possibly cached) view of memory $\sigma_i \in \mathcal{D}$. In order to formalize the consistency model, we also associate with each thread, the set of lock addresses it has acquired $\mu_i \in 2^{\mathcal{L}}$ as well as the set of addresses $w_i \in 2^{\mathcal{L}}$ whose cached values are yet to be

[1]https://github.com/goblint/analyzer.

**Table 1: Summary of notation (mnemonic aid for entire paper, not self-contained).**

| Concrete model | | Abstract analysis | |
|---|---|---|---|
| $\vec{u}\colon \mathcal{I} \to N$ | program counter for each thread | $u\colon N$ | computes invariant for each program point. |
| $\vec{\sigma}\colon \mathcal{I} \to \mathcal{D}$ | memory view of each thread | $\psi\colon N \to \mathbb{D}$ | privatized state at that program point. |
| $\vec{\mu}\colon \mathcal{I} \to 2^{\mathcal{L}}$ | locks held by a given thread. | $\lambda\colon N \to 2^{\mathcal{M}_s}$ | must-set of (symbolic) locks at that point. |
| $\vec{w}\colon \mathcal{I} \to 2^{\mathcal{L}}$ | pending writes (dirty cache) | $\Lambda\colon \mathbb{G} \to 2^{\mathcal{M}_r}$ | common (relative) locks during access. |
| $\varphi\colon \mathcal{D}$ | main memory. | $\Psi\colon \mathbb{D}$ | global invariant. |

flushed to main memory. The components of the concrete model are summarized in Table 1, which also shows the corresponding analysis abstraction of each component.

We model the execution as a non-deterministic transition system. A configuration $d_0 = (\vec{u}_0, \vec{\sigma}_0, \vec{\mu}_0, \vec{w}_0, \varphi_0)$ is an initial configuration if $\vec{u}_0$ maps each thread to an entry of a function, $\vec{\mu}_0$ maps all threads to the empty lockset, $\vec{w}_0$ states that no thread has pending cache writes, and $\sigma_0 = \varphi_0$ is a possible state of main memory after running the initialization code. Let $D_0$ denote the set of initial configurations. Given an initial state, the system evolves according to the following transitions.

At any moment, any thread $i \in \mathcal{I}$ can be chosen and the state of the system is updated. We adopt the following notational convention to describe these transitions:

$$(\vec{u}, \vec{\sigma}, \vec{\mu}, \vec{w}, \varphi) \to (\vec{u}', \vec{\sigma}', \vec{\mu}', \vec{w}', \varphi')\,.$$

We use primed symbols to denote the post-state, and we will specify only the changes to the post-state, assuming these maps remain the same for any value not explicitly mentioned in the descriptions below. There are two kinds of transitions in our system: one corresponds to the CPU issuing an instruction and the other are transitions simulating arbitrary behaviors of the memory subsystem. We first consider the latter rules.

**Flush.** Any pending writes $x \in w_i$ may be flushed to main memory such that $\varphi'(x) = \sigma_i(x)$ and $w_i' = w_i \setminus \{x\}$.

**Invalidate.** Clean locations in the cache $x \notin w_i$ may be invalidated at any time. We model this by immediately updating the value in the cache $\sigma_i'(x) = \varphi(x)$.

In real systems, and in our analysis implementation, cache invalidation only sets a flag and the cache is only updated upon an actual read request. This model is still suitable to prove the soundness of our analysis because it is a non-deterministic model that allows the refreshing of the cache at any time.

The CPU transitions will update the program counter as it issues instructions. For a thread $i$, assume there is an outgoing edge $(u_i, l, v) \in E$ from its current program counter $u_i$ to some other node $v$. If the instruction $l$ can be issued in the current configuration, we take that transition and set $u_i' = v$. We now consider the effect of issuing ordinary as well as synchronization actions. We assume an intra-thread semantic function for basic statements $[\![s]\!]\colon \mathcal{D} \to \mathcal{D}$. This may be a partial function; for example, if $s$ is a conditional guard the transition is only defined when the input state satisfies the condition of $s$. We obtain the set of read and write accesses while evaluating that instruction using the function $[\![s]\!]_a\colon \mathcal{D} \to 2^{\mathcal{L}}$ where $a \in \{r, w, rw\}$ indicates the accesses we are interested in.

**Statements.** If $[\![s]\!]\,\sigma_i$ is defined, we update the thread-local view $\sigma_i' = [\![s]\!]\,\sigma_i$. We also mark any updated locations as dirty in the cache $w_i' = w_i \cup [\![s]\!]_w\,\sigma_i$.

**Acquire.** If the lock $m$ is available, i.e., $m \notin \bigcup_j \mu_j$, we set $\mu_i' = \mu_i \cup \{m\}$ and we invalidate all clean cache locations: $\forall x \notin w_i : \sigma_i'(x) = \varphi(x)$.

**Release.** If we have the lock, i.e., $m \in \mu_i$, we release the lock $\mu_i' = \mu_i \setminus \{m\}$ and flush all pending writes: $\forall x \in w_i : \varphi'(x) = \sigma_i(x)$ and $w_i' = \emptyset$.

The side condition for acquire, which only permits the acquisition of a lock if no thread already holds it, ensures non-interleaving execution of critical sections protected by the same lock. The release transition requires that a thread holds the lock it attempts to release. According to the semantics of the Kernel mutex subsystem, attempting to reacquire a lock one already holds results in a deadlock, while attempting to release a lock one does not hold results in a failure; both cases lead to stuck states in our semantics.

*Definition 2.* The set of reachable configurations $D$ is the transitive reflexive closure of the transition relation applied to the set of initial configurations, i.e.,

$$D = \{d \mid \exists d_0 \in D_0 : d_0 \to^* d\}\,.$$

*Definition 3.* There is a race in configuration $(\vec{u}, \vec{\sigma}, \vec{\mu}, \vec{w}, \varphi)$ at location $x \in \mathcal{L}$ if distinct threads $i$ and $j$ may both issue instructions accessing that location, i.e., $(u_i, s_1, \_) \in E$ and $(u_j, s_2, \_) \in E$ with $x \in [\![s_1]\!]_{rw}\,\sigma_i \cap [\![s_2]\!]_{rw}\,\sigma_j$. We say there is a race at location $x \in \mathcal{L}$ if there exists a configuration $d \in D$ containing a race at $x$.

Here, we do not distinguish between read or write accesses. This distinction, though practically important, is not conceptually interesting and would make the subsequent analysis far more cumbersome to describe.

## 3. VALUE AND MUTEX ANALYSIS

Computing whether there exist races in a program based on the semantics described previously is infeasible. The number of interleavings grows exponentially with the number of statements even for just two threads. In what follows, our goal is to provide efficient analyses that can detect races, first assuming memory locations $\mathcal{L}$ are limited to a set of fixed integer variable names $G$, and then generalizing to abstractions of arbitrary memory locations in Section 5. The general approach is to set up a constraint system that uses *abstract* semantics instead, where the solution of the constraint system provides us reliable information about data races.

Consider the program in Fig. 2. The main thread sets the global variable $x$ to 1 and starts two threads. Globally, the shared variable $x$ has value 1. This invariant, however, is locally violated by the first thread while it holds the lock. The other thread relies on the invariant to hold whenever it acquires the lock; otherwise, it sets $x$ to an error value. Verifying that this program is free from races requires the inference of the invariant, but the invariant can only be soundly deduced if the program can be shown to be free from races. This means that the invariants and data races are to be inferred at the same time.

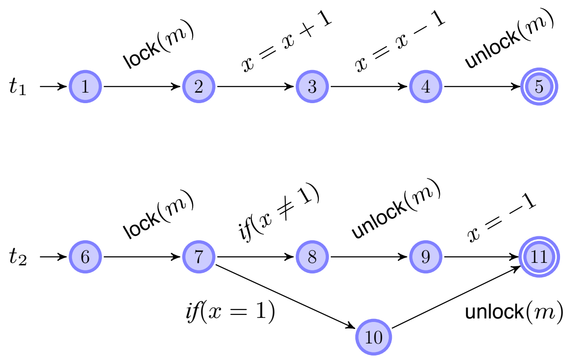

**Figure 2: Example program in our model.**

Our abstract semantics contains a global lockset mapping $\Lambda$, a global invariant $\Psi$, and for each program point, a set of locks $\lambda$ as well as a *privatized* mapping of the variables to their abstract values $\psi$. In our implementation, this mapping primarily stores local variables, but we will focus here on the framework for analyzing shared global variables that are privatized, and thus treated as local variables, within critical sections. The analysis itself is succinctly expressed as a constraint system. We will present this top-down, first showing the entire system and only then defining the functions used in the constraints. For all program points $u \in N$, edges $(u, s, v) \in E$, shared globals $x \in G$, and start points $n_t$ with $t \in \mathcal{I}$, we have the following constraints.

$$(\lambda_v, \psi_v) \sqsupseteq [\![s]\!]^\sharp(\lambda_u, \psi_u) \tag{1}$$

$$(\lambda_{n_t}, \psi_{n_t}) \sqsupseteq (\top, \psi_0) \tag{2}$$

$$\Lambda \sqsupseteq \mathsf{mf_s}(\lambda_u, \psi_u) \tag{3}$$

$$\Psi \sqsupseteq \mathsf{sync_s}(\lambda_u, \Lambda, \psi_u, \psi_u) \tag{4}$$

$$\psi_u \sqsupseteq \mathsf{sync_s}(\lambda_u, \Lambda, \psi_u, \Psi) \tag{5}$$

The first two constraints are concerned with the intra-thread behavior of the program. We assume that we have a sound value and lockset analysis that give the abstract intra-thread semantics of an individual instruction as $[\![s]\!]^\sharp$. This function is applied in the first constraint to the privatized state $\psi_u$ and the obtained values must be taken into account by the local state and lockset at the destination node. The second constraint makes sure threads start with the proper initial values. Since we track must-sets of locks, the ordering is reversed with $\top = \emptyset$ and $\bot = \mathcal{M}$; the lock-sets are initially empty.

The third constraint updates the map that specifies which globals are protected by which locks. This is done using the mutex function $\mathsf{mf_s}$ alongside interpretation of an instruction $s$ in the abstract state $\psi$. Thus, we need to constrain the lockset of the accessed variables $[\![s]\!]^\sharp_{rw}\, \psi$ with the current lockset $\lambda$.

$$\mathsf{mf_s} \colon (\mathbb{D} \times 2^{\mathcal{M}}) \to \mathbb{G} \to 2^{\mathcal{M}}$$

$$\mathsf{mf_s}(\psi, \lambda)\ x = \begin{cases} \lambda & \text{if } x \in [\![s]\!]^\sharp_{rw}\psi \\ \bot & \text{otherwise.} \end{cases}$$

As the bottom value here is $\mathcal{M}$, we effectively leave the locksets for the untouched globals alone.

The last two constraints correspond to the flushing and invalidation actions of the concrete model. Here the privatized values of unprotected globals are synchronized with the global invariant.

**Table 2: Fixpoint computation of the example.**

| $u$ | $\psi_u(x)$ | $\lambda_u$ | $\Lambda(x)$ | comment |
|---|---|---|---|---|
| 1 | 1 | $\emptyset$ | $\mathcal{M}$ | Start with $\psi_0$ and $\mathcal{M}$ |
| 2 | 1 | $\{m\}$ | $\mathcal{M}$ | lock is acquired |
| 3 | 2 | $\{m\}$ | $\{m\}$ | protected access to $x$ |
| 4 | 1 | $\{m\}$ | $\{m\}$ | $x$ is still protected |
| 5 | 1 | $\emptyset$ | $\{m\}$ | only now $\psi_5(x) \sqsubseteq \psi(x)$ |
| 6 | 1 | $\emptyset$ | $\{m\}$ | Start with $\psi_0 \sqcup \psi(x)$ |
| 7 | 1 | $\{m\}$ | $\{m\}$ | lock is acquired |
| 8 | $\bot$ | $M$ | $\{m\}$ | condition $x \neq 1$ is false |
| 9 | $\bot$ | $M$ | $\{m\}$ | dead code: not a race! |
| 10 | 1 | $\{m\}$ | $\{m\}$ | condition $x = 1$ may be true |
| 11 | 1 | $\emptyset$ | $\{m\}$ | lock is released |
| 1 | 1 | $\emptyset$ | $\{m\}$ | re-compute first thread |
| 2 | 1 | $\{m\}$ | $\{m\}$ | $\lambda_2(x) \cap \lambda(x) \neq \emptyset$ still holds |
| 3 | 2 | $\{m\}$ | $\{m\}$ | Fixpoint! |

For this, we need an operator that determines the set of globals protected by some lockset $\lambda$ according to the global lockset map $\Lambda$:

$$\mathsf{prot}_\Lambda(\lambda) = \{x \in G \mid \lambda \cap \Lambda(x) \neq \emptyset\}\,.$$

We then define a synchronization function that returns non-bottom values from a map $\psi'$ for the unprotected globals that are accessed by a statement $s$ in state $\psi$:

$$\mathsf{sync_s}(\lambda, \Lambda, \psi, \psi')\ x = \begin{cases} \psi'\ x & \text{if } x \in [\![s]\!]^\sharp_{rw}\psi \text{ and } x \notin \mathsf{prot}_\Lambda(\lambda) \\ \bot & \text{otherwise.} \end{cases}$$

Using this function, the last two constraints synchronize the privatized information with the global invariant $\psi$ on the portion of the shared state that may not be protected at a given program point.

We solve the system by fixpoint iteration. The constraint system is monotonic because as the locksets are constrained, fewer and fewer variables are privatized. We can solve the system by iterating from the least element as shown in the following example.

*Example 1.* The process of solving the constraint of the example from Fig. 2 is shown in Table 2. For each program point $u$, the values $\psi_u(x)$ and $\lambda_u$ have to be (re-)computed for a common $\Lambda(x)$. The value $\Lambda(x)$ will start off as the set of all mutexes $\mathcal{M}$ and can only decrease. The integer variable is initially unreachable $\bot$ and the lockset is the empty set. First, we compute the three values for each row going from top to bottom of the table in the natural order of the program point labels. After the first iteration we see that we must compute program points 1 and 2 again, as they have not been considered for $\Lambda(x) = \{m\}$. However, none of the respective values $\psi_u(x)$ and $\lambda_u$ nor the starting point of thread 2 will change. We have, therefore, found the least solution of the constraint system.

Having computed the least solution to the above constraint system we will flag all global variables with an empty lockset as potentially racing:

$$\mathsf{race}_\Lambda = \{x \in G \mid \Lambda(x) = \emptyset\}\,.$$

Next, we will formally relate our abstract semantics with the concrete model. This will be done using a concretization function $\gamma$ that maps abstract states $S$ into sets of concrete states $D$ that are represented by $S$. For this we require that the analysis we use also provide their concretization function — which we will also call $\gamma$. Note that the main concretization function $\gamma$ will not necessarily be monotonic. We do not require monotonicity as we will only use it to show soundness of solutions of the constraint system.

Let $S = (\lambda, \Lambda, \psi, \Psi)$ be a solution to the above constraint system. A configuration $(\vec{u}, \vec{\mu}, \vec{\sigma}, \vec{w}, \varphi) \in \gamma(S)$ iff the following conditions hold:

- For each thread $i$, whose program counter is now $u_i$, the mutexes acquired are soundly approximated by the analysis, $\lambda_{u_i} \subseteq \mu_i$, and the locksets do not overlap, i.e., $\mu_i \cap \mu_j \neq \emptyset$ only if $i = j$.
- The global invariant contains the values for all unprotected globals:
$$\forall x \notin \bigcup_{j \in \mathcal{I}} \mathsf{prot}_\Lambda(\lambda_{u_j}) : \varphi(x) \in \gamma(\Psi(x)) .$$
- The way thread $i$ views memory is correctly represented for all globals it may access or for which it has pending writes:
$$\forall x \in \mathsf{prot}_\Lambda(\lambda_{u_i}) \cup \mathsf{race}_\Lambda \cup w_i : \sigma_i(x) \in \gamma(\psi_{u_i}(x)) .$$
- The set of pending writes $w_i$ must include all updated protected globals:
$$\{x \in \mathsf{prot}_\Lambda(\lambda_{u_i}) \mid \sigma_i(x) \neq \varphi(x)\} \subseteq w_i .$$

The analysis does not care about values that cannot be accessed by the given threads. Also, the global invariant only partially specifies the values of main memory. Within a critical section, the state of main memory could have any value as this does not influence the value of the computation after synchronization points. Thus, we may include some configurations in our interpretation of the analysis result that are not truly reachable, but what really matters is that we are certain to include all reachable configurations.

Theorem 1 (Soundness). *If $S$ is the least solution to the above constraint system and $D$ is the set of reaching configurations, we have $D \subseteq \gamma(S)$.*

Proof idea. As $D$ is the least set closed under the transitions defined in the previous section, we would need to show that the concretization of our analysis results $\gamma(S)$ is also closed under each of the transition rules. We will briefly describe the key observations that a formal soundness proof would rely on. For ordinary instructions, constraints (3) and (5) ensure that all values accessible by a thread at program point $u$ are soundly over-approximated in the privatized state $\psi_u$. The resulting computation, therefore, remains in $\gamma(S)$ as long as our intra-thread transfer functions are sound.

Let us also consider a release action. In this case, a previously protected location may now be unprotected and other threads may potentially access it. We did not require protected values to be represented by our global invariant, but the value is correctly represented in the local state $\psi_u$. As we require that protected locations with differing cached values mark such locations as dirty, the outstanding write will complete, and by constraint (4) that value is propagated into the global invariant. Conversely, during an acquire action, some global $x$ may now be protected by a thread $i$. In that case, the value of $x$ is either already represented in $\psi_u$, or we have $x \notin w_i$, so the cache must be invalidated, and by constraint (5), the local state will take the correct value of $x$ from the global invariant. □

Theorem 2 (Race freedom). *Let $S = (\lambda, \Lambda, \psi, \Psi)$ be a solution to the constraint system. For any global $x \in G$, if there is a race at $x$, we have $x \in \mathsf{race}_\Lambda$.*

Proof. For there to be a race at variable $x$ there would need to be some reachable configuration where two different threads both issue instructions accessing $x$. Given that their locksets may not overlap and our analysis is sound, $\Lambda(x)$ would be constrained by two sets with no elements in common, ensuring $\Lambda(x) \subseteq \emptyset$. □

The analysis can be instantiated with different abstract domains and our framework allow these analyses to be both path- and context-sensitive. Path-sensitivity is important in order to deal with conditional locking or potentially failing locking operations. For this, we use a property-simulation abstract domain [12]. The relevant property is the set of definitely held locks, so we never join locksets; instead, we track the value abstraction for all possible locksets. In this way, an operation like $r = \mathsf{try\_lock}(m)$ will split the state, reflecting in the return value $r$ whether the lock is acquired or not.

We have not included function calls and local variables in this framework because our treatment is fairly standard and relies on the entry and combine operators used in the functional approach to inter-procedural analysis [41]. Applying the flow-insensitive treatment of shared globals in a context-sensitive setting requires computing partial global invariants [39], as we only want consider the contexts that are encountered during analysis, resulting in side-effecting constraint systems [3].

## 4. A GENERAL APPROACH

We now extend the notion of a memory location from a set of static variables to static identifiers that represent disjoint portions of shared memory locations. We do not rely on a fixed heap abstraction; instead, our goal is to arrive at a flexible framework that combines different analyses. We may even take a more Einsteinian view of space and encode temporal information as well. Generally speaking, race freedom is a conditional proposition of the form:

"If two accesses *may* conflict,
certain safety conditions *must* hold."

For example, if two access expressions may alias, their corresponding lock expressions must alias. There are, however, other justifications for excluding races that fit within this paradigm. We let each analysis contribute to either side of the conditional: a thread uniqueness analysis provides an additional safety guarantee excluding races, while a happens-before analysis can refine the notion of when accesses may conflict.

Instead of the set $G$ of variables from a fixed set of names, we now move to an abstract domain $\mathbb{G}$, representing mainly may-alias equivalence classes. Combining may-alias information is straightforward: given two analyses that soundly partition memory into may-alias equivalent classes, their Cartesian product will also form a sound partitioning. Our privatization framework requires that we not only answer pairwise may-alias queries, but that we give a canonical representative for each address expression. This is provided by, e.g., by region-based shape analyses [19, 38]. We refer to the canonical representative of a heap region as the *owner* of that region. In order to correlate regions with locks, owners must refer to unique memory locations that can be described statically. Such descriptions, however, may depend on parameters, such as the base address and integer offsets, to specify concrete locations in the heap.

*Example 2.* Consider the following example of per-element locking. The assumption here is that the parameter is entirely unknown to us; still, it should not be hard to establish that the program is race-free for any element of type $\tau_{node}$.

```
struct node { mutex mtx; int data; }
thread t(struct node *p) {
  lock(&p→mtx);
  p→data++;
  unlock(&p→mtx);
}
```

We have no better description of the region this access belongs than its type and the field that was accessed. For the lock, we have to keep a symbolic lockset $\{p{\to}mtx\}$, and if at the time of access, we can establish that the base pointers are equal, we convert the symbolic lockset to a relative lockset $\{\star\,\texttt{.}\,mtx\}$ by replacing references to real variables with symbolic host variables. For this example, we get

$$\Lambda(\tau_{node}\,\texttt{.}\,data) = \{\star\,\texttt{.}\,mtx\}\,.$$

This will serve to decouple our may- and must-alias information, so that we can map an arbitrary may-alias equivalence class to a lockset and use $\star$ to refer to individual elements of this equivalence class.

*Example 3.* This idea can also be applied to deal with array-based locking. If we have an array of locks that is used statically, we could simply add concrete indexes to the lockset. If we can establish an equality, though, we replace the index with a symbolic index. Consider a simple program now with an array of locks protecting an array of data:

```
mutex mtxs[1024];    thread t1(int i) {
int   data[1024];      lock(&mtxs[i]);
                       data[i]++;
                       unlock(&mtxs[i])
                     }
```

We will now replace anything established an equal by an integer must-equality analysis with symbolic indices:

$$\Lambda(data[\top]) = \{mtxs[\star_0]\}\,.$$

*Example 4.* We say that a thread is unique if only a single instance of a given thread template is ever spawned. We can assume that a unique thread $t$ always holds a mutex $m_t$. There is a classic example in the Intel Thread Checker tutorial where the thread created by parameter $i$ is the only thread that accesses element $data[i]$.

```
int data[1024];        int main() {
                         int i = 0;
thread t(int i) {        while (i < 1024)
  data[i]++;               spawn(t, i);
}                      }
```

Given an analysis that ensures $i$ takes unique values in each iteration of the while loop, we can verify this example by reducing it the case above where the array is protected by an array of locks:

$$\Lambda(data[\top]) = \{m_t[\star_0]\}\,.$$

*Example 5.* A similar approach enables us to make use of domain-specific knowledge. Consider following two functions:

```
void open(file *f){    void close(file *f){
  g++;                   g--;
  f→data++;              f→data--;
}                      }
```

Assume that there is an environmental guarantee that for each file $f$, these functions are called sequentially with open always preceding a close; however, different files may be opened concurrently. Based

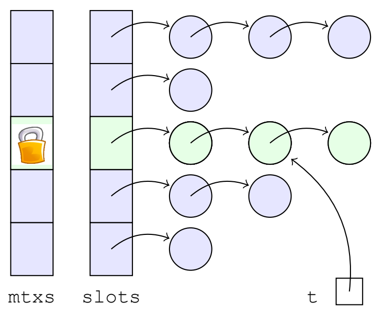

**Figure 3: Medium-grained locking scheme [38].**

on this information, we are able to deduce that each access to $f$ in function *open* is protected by the lock $m_{open}[\star_0]$ and that accesses to $f{\to}data$ in *open* and *close* cannot race as they are time-wise disjoint:

$$\Lambda(\tau_{file}\,\texttt{.}\,data,\ open) = \{m_{open}[\star_0]\}$$
$$\Lambda(\tau_{file}\,\texttt{.}\,data,\ close) = \{m_{close}[\star_0]\}$$

We need the uniqueness mutex to express that there can be no concurrent opens and the time-regions to express that open and close may not happen concurrently. Note that no such guarantees are extended to the global variable $g$, so we have $\Lambda(g, \top) = \emptyset$, meaning that the access can happen in any time period.

*Example 6.* Let us now consider dynamically allocated structures. We can extend per-element locking just as easily to allocation-site abstraction, but we now need to consider locks that protect an entire data structure, such as a linked list. As our algorithm is parametric on how these may-alias equivalence classes are obtained, we do not need to care if we are dealing with may-points-to sets or the result of a more sophisticated heap analysis. We do, however, need the heap analysis to export array index expressions.

Consider now a synchronized hash table (Figure 3). When the pointer $t$ starts to traverse a certain bucket, our pointer analysis may be able to say that this belongs to the region described by $slots[i]$.

```
lock(&mtxs[i]);
t = slots[i]→next;
while (t != null) {
  t→data++;
  t = t→next;
}
unlock(&mtxs[i]);
```

We may then note that the $i$ used when $t$ first entered the memory region $slots[i]$ is equal to the index used when acquiring the lock $locks[i]$. If a region analysis can determine when these buckets are disjoint, we can essentially reduce this case to the previous example:

$$\Lambda(R(slots[\top]), \tau_{ln}\,\texttt{.}\,data) = \{mtxs[\star_0]\}$$

To incorporate this analysis into our framework, the analysis provides the symbolic index expression of the region that a pointer belongs to. When we access $t{\to}data$, the region analysis knows that $t \in R(slots[i])$ and the index expression $i$ must be made available to our framework, so we can make sure it corresponds to the index used when acquiring a lock.

*Example 7.* The following example, although contrived, will clarify the intuition behind our approach. Assume that the data field of our struct is itself an array and we now have a matrix of locks:

```
lock(&mtxs[i][0]);
  t = slots[i]→next;
  lock(&mtxs[i][j+1]);
    t→data[j]++;
  unlock(&mtxs[i][j+1]);
unlock(&mtxs[i][0]);
```

The first lock of each row is reserved for the slots, so the rest of the locks are offset by one. It is worth seeing how this is expressed in our locksets:

$$\Lambda(R(\mathit{slots}[\top]),\ \tau_{\mathit{ln}}\,\mathbf{.}\,\mathit{next}) = \{\mathit{mtxs}[\star_0][0]\}$$
$$\Lambda(R(\mathit{slots}[\top]),\ \tau_{\mathit{ln}}\,\mathbf{.}\,\mathit{data}[\top]) = \{\mathit{mtxs}[\star_0][\star_1 + 1]\}$$

When we access $t{\to}\mathit{data}[j]$, we are given the list of index expressions $[i, j]$, which we use to check against the expressions used when acquiring the locks. Our framework does not know that $i$ is part of the region and $j$ is the offset of the access, but relies on whatever combination of may-alias analyses we use to be consistent (in a way made precise in the next section) whenever asked for the list of index expressions.

*Example 8.* Finally, we consider our introductory example from Figure 1, where we have a region of memory that protects the lock field embedded into elements whose data fields are protected by their own mutexes.

```
struct node { int data; int mtx;
              struct list_head list; }
```

Assume we have a global mutex `list_lock` and a global list of nodes `ndlist`. We may be iterating through `ndlist`, attempting to find the last node having a given value:

```
struct node *find(int id){
  struct list_head *lp = ndlist.next;
  struct node *np, *res;
  lock(&list_lock);
  while (lp != &ndlist) {
    np = container_of(lp, list);
    lock(&np→mtx);
    if (np→data == id) res = np;
    unlock(&np→mtx);
    lp = lp→next
  }
  unlock(&list_lock);
  return res;
}
```

In this case, we will have the following common locksets while executing the find function:

$$\Lambda(R(\mathit{ndlist}),\ \tau_{\mathit{lh}}\,\mathbf{.}\,\mathit{next}) = \{\mathit{list\_lock}\}$$
$$\Lambda(R(\mathit{ndlist}),\ \tau_{\mathit{node}}\,\mathbf{.}\,\mathit{data}) \subseteq \{\mathit{list\_lock}, \star\,\mathbf{.}\,\mathit{mtx}\}$$

The above function returns the found node, which the user may access without the list lock, restricting the lockset further:

$$\Lambda(R(\mathit{ndlist}),\ \tau_{\mathit{node}}\,\mathbf{.}\,\mathit{data}) = \{\star\,\mathbf{.}\,\mathit{mtx}\}$$

The access to the list fields, however, is not influenced by this access because both the field and the type for that access are different.

For this example, we did not actually use the region information, but the moment we have another list using the kernel API list type $\tau_{\mathit{lh}}$, we would be accessing the same fields for all lists in the program. If we want to verify cases where different lists are protected by different locks, we need this combination of region analysis with field-sensitivity and type information.

## 5. SYMBOLIC AND RELATIVE LOCKS

We now formalize the ideas from the previous section. The set $\mathbb{G}$, a product of individual partitioning domains, serves as a more general notion of globals. Our fundamental assumption is that the heap analyses are such that whenever we have some abstraction $\psi$ of a concrete heap $\varphi$, we can conceptually map a concrete memory location $l$ to a unique owner and a list of index values $\mathit{own}(\varphi, \psi)\ l \in \mathbb{G} \times \mathbb{N}^*$. While we use these indices to analyze races, our value analysis is not sensitive to indices; i.e., we compute a single abstract value that must over-approximate the entire region.

The critical change to our lockset analysis is rather the sets we track for each program point: we now track *symbolic locks* $\mathcal{M}_s$: an expression of the form $\mathsf{lock}(\mathit{adr})$ will thus add the expression $\mathit{adr}$ to the symbolic lockset. For an unlock of $\mathit{adr}'$, we must remove all locks that may possibly alias with $\mathit{adr}'$. The syntax for address expressions we consider is as follows:

$$\mathit{adr} \ ::= \ \&A \mid p \mid \mathit{adr}\,\mathbf{.}\,f \mid \mathit{adr}\,\mathbf{.}\,[e]$$

where $f$ is a field selector and $e$ is an index expression. In C syntax, the left-associative infix operator "**.**" is defined by:

$$\mathit{adr}\,\mathbf{.}\,f = \texttt{\&adr}{\to}\texttt{f} \qquad \mathit{adr}\,\mathbf{.}\,[i] = \texttt{\&(*adr)[e]}$$

Note that the prefix address-of operator has a lower priority than the postfix operators. The above expressions are thus purely address computations where we apply offsets to a base pointer; no pointer is dereferenced during evaluation.

*Example 9.* The following is the C translation of two example address expressions:

$$(\&A)\,\mathbf{.}f\mathbf{.}\,[e]\,\mathbf{.}\,g \qquad \texttt{\&A.f[e].g}$$
$$p\,\mathbf{.}f\mathbf{.}\,[e]\,\mathbf{.}\,g \qquad \texttt{\&p}{\to}\texttt{f[e].g}$$

In the case of actual dereferences our front end introduces temporary variables; for example, the expression `lock(&p→f→mtx)` is translated into the statements `t = p→f; lock(&t→mtx)`. Our symbolic lockset analysis adds $t\,\mathbf{.}\,\mathit{mtx}$ to the lockset and relies on our must-equality analysis [40] to track the equality $t \equiv p{\to}f$. We can then use that equality to associate an access `p→f→data` with the symbolic lock $t\,\mathbf{.}\,\mathit{mtx}$; however, we do not add `p→f→mtx` to our symbolic lockset because this address has not a constant offset to its base pointer $p$.

Recall that the key mapping in our race detection algorithm was $\Lambda\colon G \to 2^{\mathcal{M}}$, which maps a shared variable to the set of locks always held while accessing that shared variable. The notion of a shared variable is now expanded to these owners in $\mathbb{G}$. We will again map each owner to a set of locks, but our symbolic locksets may refer to local variables. For the global mapping of always held locks, we interpret the locks relative to its owner. A relative lock is an address expression just like our symbolic locks, except the base pointer may be either a global variable or a designated host variable:

$$\mathit{adr} \ ::= \ \&A \mid \star \mid \mathit{adr}\,\mathbf{.}\,f \mid \mathit{adr}\,\mathbf{.}\,[e_\star]$$

Similarly, integer expressions $e_\star$ may not refer to local integer variables but instead use special variables $\star_0, \star_1, \ldots$ when lock expressions and access expressions have matching indices, such as in Example 3.

In order to apply the privatization-based race detection framework from in Section 3, we need a function $\mathsf{rel}$ that translates symbolic locksets to relative locksets, using pointer and integer equality information. Assuming the soundness of this translation, the special constants are treated as uninterpreted symbols and do not differ from concrete names. Although the structure of shared owners $x \in \mathbb{G}$ is opaque to us, we will require the access function provided by the domain $[\![s]\!]_{rw}$ to now return a tuple of the form $(x, es)$, which contains additionally the list of relevant expressions.

We now discuss the modifications to the auxiliary functions used to constrain $\Lambda$. At the time of an access, we have symbolic locksets that contain references to the current state of local variables. These values do not make sense globally, so the function giving us the set of commonly held locks must perform a translation. We will again present definitions top-down, using operations that will be defined after their use. We first give the redefined the mutex map function for accessed variables:

$$\mathsf{mf_s}\colon (\mathbb{D} \times 2^{\mathcal{M}_s}) \to \mathbb{G} \to 2^{\mathcal{M}_r}$$

$$\mathsf{mf_s}(\psi, \lambda)\ x = \begin{cases} \mathsf{rel}(\psi, es, \lambda) & \text{if } (x, es) \in [\![s]\!]^\sharp_{rw}\,\psi \\ \bot & \text{otherwise} \end{cases}$$

The only difference from Section 3 is the use of the translation function rel that performs the translation from symbolic to relative locksets:

$$\mathsf{rel}\colon (\mathbb{D} \times E^* \times 2^{\mathcal{M}_s}) \to 2^{\mathcal{M}_r}$$

$$\mathsf{rel}(\psi, es, M) = \{\, m' \in \mathcal{M}_r \mid \exists m \in M : [es, \vec{\star}]_\psi\ m\ =\ m' \,\}$$

The semantic substitution operator $[x, r]_\psi\ e$ replaces every subexpression $e'$ in $e$ with $r$ whenever $\psi \models e' \equiv x$. Just as with syntactic substitutions, we can lift this definition to a pair of lists by composing the individual substitutions corresponding to each element of the zipped list:

$$[[a_1, \ldots, a_n], [x_1, \ldots, x_n]]_\psi\ = [a_n, x_n]_\psi \circ \cdots \circ [a_1, x_1]_\psi$$

Intuitively, we apply the semantic substitution to each element of the relative locks and keep only those where all references to local variables have been substituted. Note that this is sound as long as our equality analysis is sound, but we may fail to establish an equality, in which case our analysis will flag this access as unprotected.

The synchronization function will also have to convert symbolic locksets to relative ones in order to check the intersection with the set of locks that are always held when accessing that shared region:

$$\mathsf{sync_s}(\lambda, \Lambda, \psi, \psi')\ g =$$
$$\begin{cases} \psi'\ x & \text{if } (x, es) \in [\![s]\!]^\sharp_{rw}\psi \text{ and } g \notin \mathsf{prot}_\Lambda(\mathsf{rel}(\psi_i, es, \lambda)) \\ \bot & \text{otherwise} \end{cases}$$

With these modifications in place, the constraint system remains exactly the same as before. We will now give some intuition about the semantics of privatization in the context of relative locksets. This is best appreciated by considering an example of per-element privatization.

*Example 10.* We will again use a simplistic integer example of an invariant being temporarily violated within a critical section.

```
for (i = 0; i++; i < 100) {
  lock(&mtxs[i]);
  data[i]++;  // invariant violated
  data[i]--;  // invariant restored
  unlock(&mtxs[i]);
}
```

Assume five different threads execute this code. Then, at a given moment, each thread might have privatized a specific index of the array. We now describe the meaning of our analysis results for such programs.

In order to relate the invariant-based semantics using symbolic locksets with the concrete model, our concretization must now be able to handle symbolic locksets. First, symbolic locksets must be subsumed by the concrete lockset after being evaluated in the corresponding concrete state, i.e., $\forall i \in \mathcal{I} : [\![\lambda_{u_i}]\!]\,\sigma_i \subseteq \mu_i$. If we look back at the definition in Section 3, we must now rewrite the concretization conditions to allow arbitrary locations instead of the fixed names. Mainly, we need to adapt $\mathsf{prot}_\Lambda(\lambda)$ to this new setting. Our memory abstractions are such that conceptually each concrete memory location corresponds to a pair $(x, is) \in \mathbb{G} \times \mathbb{N}^*$. We must determine whether these specific indices *is* of region $x$ are protected in the concrete state $\sigma$ by the symbolic locks $\lambda$. For this, we check whether the indices match a symbolic lockset if variables are replaced with their concrete values from $\sigma$. We can overload the function rel for this, keeping in mind that we are now relativizing locks in a concrete setting to give meaning to our analysis results. We thus determine whether a location is protected using $\mathsf{prot}_\Lambda(\mathsf{rel}(\sigma, is, \lambda))$. This is effectively the only change needed: recall that we compute a single invariant for each region and only allow indices to be privatized within critical sections.

Theorem 3 (Heap Soundness). *Given a heap abstraction that relates memory locations $l \in \mathcal{L}$ to indexed regions, we have $D \subseteq \gamma(S)$ also for heap-manipulating programs.*

Proof idea. The main threat to soundness, compared to the simple case, is to ensure that distinct locks *mtx*[3] and *mtx*[7] are not relativized into the same relative lock *mtx*[$\star_1$] when accessing the same location. Assume we have two accesses to the same region with index expressions $es_1$ and $es_2$ in states $\sigma_1$ and $\sigma_2$, holding symbolic locks $m_1$ and $m_2$ that were relativized to the same lock. If the index expressions coincide, we have $[\![es_1]\!]\,\sigma_1 = [\![es_2]\!]\,\sigma_2$. This equality, together with the fact that relative mutexes have had all their variables successfully substituted, allows us to make the critical inference:

$$[es_1, \vec{\star}]_{\sigma_1}\, m_1 = [es_2, \vec{\star}]_{\sigma_2}\, m_2 \implies [\![m_1]\!]\,\sigma_1 = [\![m_2]\!]\,\sigma_2$$

That is, relative locks coincide when accessing the same location only if the corresponding concrete locks coincide. This is essentially the only additional idea in our proofs of soundness and race freedom. $\square$

## 6. EXPERIMENTAL EVALUATION

Our benchmark suite consists of 26 device drivers — all character device drivers of the 4.0 version of the Linux kernel that CIL [33], the front-end library for Goblint, can digest. The size of the programs in the suite range from 96 to 3184 lines of Physical Source Lines of Code (SLOC) generated using David A. Wheeler's 'SLOCCount'. The performed analysis integrates value, points-to, (symbolic) lockset, region, thread (uniqueness), and symbolic equality analyses — as described in the previous section. Additionally, the analysis was configured to be path- and context-sensitive [2, 45] to increase precision.

A high-level overview of the analysis results is shown in Table 3. For each analyzed driver we note the analysis time, number of memory locations found safe, and two categories of potentially unsafe locations. Ten drivers were analyzed in under one second each, 13 drivers were analyzed in between one and ten seconds each, and the three remaining drivers required ten to about 75 seconds each. Our statistics is based on memory locations, rather than access sites or pairs of accesses, which may seem more natural measures. This is

**Table 3: Overview of analysis results for each driver**

| Driver | Time | Safe | Race | |
|---|---|---|---|---|
| | | | Indirect | Direct |
| hangcheck-timer.c | 0.5 s | 6 | 12 | 0 |
| mem.c | 0.8 s | 28 | 0 | 5 |
| dtlk.c | 3.1 s | 12 | 29 | 19 |
| efirtc.c | 0.8 s | 106 | 0 | 0 |
| genrtc.c | 0.9 s | 107 | 0 | 2 |
| lp.c | 1.1 s | 21 | 84 | 5 |
| toshiba.c | 1.1 s | 111 | 0 | 0 |
| nvram.c | 0.8 s | 111 | 0 | 1 |
| misc.c | 0.5 s | 120 | 6 | 10 |
| applicom.c | 2.3 s | 134 | 14 | 8 |
| ipmi_poweroff.c | 1.1 s | 6 | 142 | 11 |
| random.c | 73.6 s | 97 | 65 | 31 |
| rtc.c | 1.0 s | 170 | 20 | 4 |
| scx200_gpio.c | 0.4 s | 5 | 188 | 1 |
| ttyprintk.c | 0.6 s | 2 | 193 | 3 |
| apm-emulation.c | 1.2 s | 178 | 29 | 12 |
| ppdev.c | 1.3 s | 119 | 129 | 14 |
| raw.c | 0.9 s | 18 | 264 | 3 |
| pc8736x_gpio.c | 1.0 s | 112 | 195 | 2 |
| ipmi_watchdog.c | 12.9 s | 63 | 245 | 13 |
| hpet.c | 2.9 s | 59 | 292 | 22 |
| tlclk.c | 0.9 s | 299 | 88 | 6 |
| ipmi_devintf.c | 3.4 s | 247 | 196 | 8 |
| ipmi_msghandler.c | 26.0 s | 0 | 460 | 98 |
| bsr.c | 1.0 s | 592 | 19 | 10 |
| sonypi.c | 8.7 s | 276 | 350 | 27 |

because our analysis framework tracks abstract values associated with each abstract memory location, so even a single unprotected access means that we may not privatize that region of memory. We therefore count the location as unprotected, although all other accesses to that abstract location may be properly synchronized. The abstract description of a memory location depends on the precise analyses use. We therefore report races using a standard allocation site abstraction of the heap. Although the analyzer may use finer heap abstractions to distinguish accesses to different objects allocated at the same site, these are all grouped for our statistics to make the results comparable when enabling different heap analyses.

A potential race is considered *direct* if it is seen as unsafe based on direct accesses, i.e., when a location is accessed directly by an expression in the source code, e.g., by an assignment to the location. An *indirect* race, on the other hand, is reported when safety cannot be guaranteed when we additionally consider indirect accesses. An indirect access can happen if a pointer to the location is potentially passed to an `extern` function — a function for which the analyzer does not have the source code.

One source of indirect accesses comes from the use of various container data structures, where, for example, the API call that inserts an object into the container may theoretically access all objects in the container. Another example of indirect accesses is the `private_data` field in the `file` structure of Linux. In that use-case, pointers to the driver's private data are inserted into the structure and passed around with it. That means that data race freedom is guaranteed only by convention.

Direct races reported here are either real data races or false positives that generally arise due to the use of unsupported protection mechanisms, additional assumptions on the environment, or simply

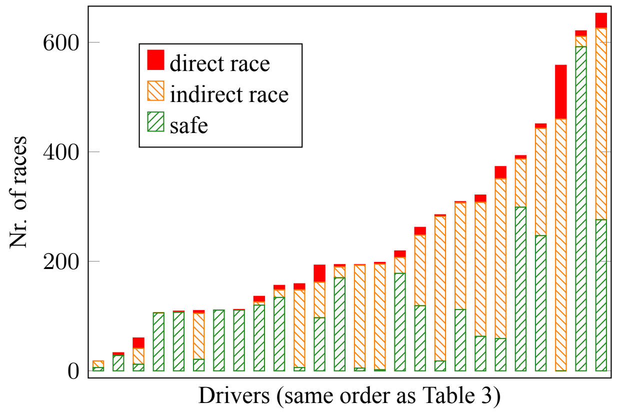

**Figure 4: Distribution of direct and indirect races**

analysis precision loss. Proportions of safe locations versus direct and indirect races can be seen in Fig. 4. On average 5% of locations are reported as direct races — we count this as a success. The number would even drop to 3.3% if the outliers `random.c` and `ipmi_msghandler.c` were removed. These two differ from rest of the drivers in the benchmark suit in either their assumptions or code complexity. The first, `random.c`, does not expose most of its non-static functions outside the Linux kernel. Instead, most of the functions are used within the kernel code to generate entropy. It seems that in the majority of the locations where Goblint detects a data-race, the responsibility for thread safety is left to the caller. As the computation of entropy is approximate and security only requires lower bounds on this value, it is likely that such races are benign and at worst only fail to count added entropy. Increasing precision, in this case, would require more domain-specific information about the environment in which the functions are used.

The module `ipmi_msghandler.c` has a higher complexity than most other character device drivers. It uses advanced patterns that Goblint does not yet support, e.g., Read-Copy-Update, atomic read and update operations, and per-element locking where the lock has a non-constant offset to its base pointer. Locking patterns that Goblint can follow are not adhered to in “unregister”, “panic”, and “initialization” modes, where running initialization several times is avoided by a boolean flag whose safety is unclear. In addition to that, the usage of variables suggests that some of its input are assumed to be not shared and should not be accessed at the same time. Effective analysis of this driver requires more effort to handle the advanced patterns and integration of information about the environment it is used in.

The number of indirect races is on average much larger than the number of direct races. In fact, on average 47% out of all locations are ruled indirect races. Fourteen of the 26 drivers had a significant portion of the memory locations escaped. These issues we count as a failure of our tool in its current form, but not necessarily failure of the approach, as the safety of API functions is not automatically derivable. One solution would be manual annotations.

Next, we evaluate the benefit that adding advanced features to Goblint has brought. Table 4 shows data for all modules that were affected by switching the specific features on and off. Out of the 26 drivers, 19 use only static locks and seven additionally use some relative locking pattern. The addition of symbolic locking is able to decrease the number of reported races in only `ipmi_msghandler.c`. But additionally there are several near misses. In `lp.c`, the exported function `lp_release` does not acquire the relative lock

**Table 4: Contributions of features**

| Driver | Direct race count for feature set | | | |
|---|---|---|---|---|
| | Base | +Region | +Symb. | +Both |
| apm-emulation.c | 15 | 12 | 15 | 12 |
| hpet.c | 26 | 22 | 26 | 22 |
| ipmi_devintf.c | 10 | 8 | 10 | 8 |
| ipmi_msghandler.c | 140 | 114 | 127 | 89 |
| ppdev.c | 20 | 14 | 20 | 14 |
| random.c | 38 | 37 | 33 | 31 |

`lp_table[minor].port_mutex`, contrary to the pattern established by other functions — this would make three memory locations with direct races safe. Relative locking, we conclude, is used in complicated drivers and needs to be supported, but its analysis practically requires taking environmental assumptions into account. A slightly better situation is with dynamic lists — six drivers use dynamic lists and five of them get fewer warnings when enabling region analysis. But also there the results could be better if more environmental assumptions could be discovered and used in the analysis.

We conclude that flexible, generic combination of existing analyses can be decently successful for analyzing data races in open programs. To reach this level, we needed to pay close attention to precision when implementing support for popular locking patterns. At this point, however, the limiting factor is the environment of the module. Adding more and more specific features will have a minor effect if safety hinges on the convention of use of the module. A similar observation was made by Logozzo et al. [27], who suggest inferring assumptions as preconditions and analyze subsequent versions of the software under those assumptions. We would need assumptions about whether certain threads may run in parallel.

## 7. RELATED WORK

We will focus here on automated static approaches to race detection. The Astreé analyzer [11] has in recent years been extended to analyze concurrent programs [31]. The concurrency abstraction is based on threads propagating *influences* to other threads [29]; effectively, this computes a flow-insensitive invariant for the global state very similar to our privatization framework and is proven sound with respect to the reordering of commands that are allowed in weak memory models. Miné [30] further extends the framework to take relational invariants into account. We did not propagate relational invariants in this paper, but these are straightforward in our setting with an explicit global invariant. Astreé ensures the absence of all runtime errors and also detects race conditions, but it is not clear how they deal with dynamically allocated locks; in their formal expositions, a finite set of locks is assumed.

Whoop [13] is a recent race detection tool that combines race detection with a precise bug detector to eliminate false alarms. The static race detector works by instrumenting the program such that symbolic execution can generate the proof obligations needed to verify race freedom. It does not consider dynamically allocated locks. Other race detectors, including Locksmith [34], CoBE [24], and Relay [46], focus only on aliasing analyses, ignoring abstraction of integer variables. We share some similarity with these approaches. The composition of may-alias analysis is exploited in CoBE to speed up pointer alias analyses by running them in a sequence from more coarse-grained to more fine-grained Kahlon [22]. Relay uses symbolic locksets for procedure summarization. They also use the term "relative locksets" to summarize the effect of a function on the set of definitely held locks; that is not related our use of the term.

Compared to state-of-the-art race detection tools for C, our approach has the following advantages. We can handle both value-dependent synchronization patterns and consider dynamically allocated locks. In particular, we handle per-element locking without the need for programmer annotations. The Locksmith analyzer uses existential types for this, but relies on programmer annotations [35]. We can also handle medium-grained locking, such as a linked list being protected by a list lock. This feature is unique to Goblint among all race detection tools we have tested, including commercial tools. Last, but certainly not least, we can analyze locking schemes involving arrays of locks, as may occur in a synchronized hash table. For Java, Naik and Aiken [32] propose *conditional must-not aliasing* to deal with locking schemes of various levels of granularity.

The analysis of heap-manipulating multi-threaded programs is an active field of research in its own right. Most concurrent shape analyses algorithms, e.g. [4, 9, 28, 44], focus primarily on proving properties such as memory safety, data structure invariants, and linearizability for small but complex concurrent implementations of data structures. The basic idea of our approach resembles the thread-modular shape analysis of Gotsman et al. [17]. That analysis, however, relies on a-priori race-detection and lock set computation to enable sequential shape analysis of concurrent programs. In contrast, we jointly analyze values and establish race freedom. More lightweight pointer analyses have also been developed that also combine with integer values [5, 10, 16, 18, 36].

For static device driver verification, the method of choice seems to be software model checkers, where SLAM [6] has made its way into the Windows Driver Framework, and for Linux, the Linux Driver Verification (LDV) project have extended the BLAST model checker [7] with pointer analyses to analyze device driver code [42]. From our perspective, the most relevant work are methods for model checking concurrent programs [20, 21, 43], but the focus so far has not been on detecting race conditions and sequential consistency is assumed. The LDV project reports on developing a race detection tool on top of the CPAChecker framework [8]. This is an extensible framework that combines model checking and static analysis, featuring automatic abstract domain selection [1]. Ideas from this paper could be integrated into that framework, allowing more expensive analyses to be only used for complicated idioms.

## 8. CONCLUSION

We have presented the analysis framework Goblint. The concept of *global invariants* refined by a notion of *privatization* enables us to realize reasonably precise yet scalable analyzers of multi-threaded C programs. We applied this approach to realize precise and efficient data race analyzers. On top of a generalization of lock set analysis, dedicated analyses are provided which cover specific programming idioms such as value-dependent locking or per-element locking. We argue that, due to the flexibility of the general framework, additional analyses can easily be provided to catch further locking patterns; however, inferring environmental constraints may now be the most promising way forward.

## 9. ACKNOWLEDGEMENTS

This work was supported by institutional research grant IUT2-1 from the Estonian Research Council and ITEA 3 Project ASSUME ("Affordable Safe & Secure Mobility Evolution"). We are grateful to the anonymous reviewers for many suggested improvements and some very hard questions.

## 10. REFERENCES

[1] S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. v. Rhein. Domain Types: Abstract-Domain Selection Based on Variable Usage. In *Hardware and Software: Verification and Testing*, pages 262–278. LNCS 8244, Springer, 2013.

[2] K. Apinis. *Frameworks for analyzing multi-threaded C*. PhD thesis, Institut für Informatik, Technische Universität München, June 2014.

[3] K. Apinis, H. Seidl, and V. Vojdani. Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In *APLAS'12*, pages 157–172. LNCS 7705, Springer, 2012.

[4] H. Attiya, G. Ramalingam, and N. Rinetzky. Sequential verification of serializability. In *POPL'10*, pages 31–42. ACM Press, 2010.

[5] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *SAS'06*, volume 4134 of *LNCS*, pages 221–239. Springer, 2006.

[6] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *POPL'02*, pages 1–3. ACM Press, 2002.

[7] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[8] D. Beyer, T. A. Henzinger, and G. Theoduloz. Program Analysis with Dynamic Precision Adjustment. In *ASE'08*, pages 29–38, 2008. .

[9] C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive resource invariant synthesis. In *APLAS'09*, volume 5904 of *LNCS*, pages 259–274. Springer, 2009.

[10] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL'08*, pages 247–260. ACM Press, 2008.

[11] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP'05*, LNCS 3444, pages 21–30. Springer, 2005.

[12] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI'02*, pages 57–68. ACM Press, 2002.

[13] P. Deligiannis, A. F. Donaldson, and Z. Rakamarić. Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers. In *ASE'15*, pages 166–177, Washington, DC, USA, 2015. IEEE Computer Society.

[14] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP'03*, pages 237–252. ACM Press, 2003.

[15] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *ISCA'90*, pages 15–26. ACM, 1990.

[16] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *POPL'05*, pages 338–350. ACM Press, 2005.

[17] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI'07*, pages 266–277. ACM Press, 2007.

[18] S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *POPL'09*, pages 239–251. ACM Press, 2009.

[19] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL'05*, pages 310–323. ACM Press, 2005.

[20] T. A. Henzinger, R. Jhala, and R. Majumdar. Race Checking by Context Inference. In *POPL'04*, pages 1–13. ACM Press, 2004.

[21] O. Inverso, T. L. Nguyen, B. Fischer, S. L. Torre, and G. Parlato. Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-threaded C-Programs. In *ASE'15*, pages 807–812, 2015.

[22] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI'08*, pages 249–259. ACM Press, 2008.

[23] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV'07*, volume 4590 of *LNCS*, pages 226–239. Springer, 2007.

[24] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *ESEC/FSE'09*, pages 13–22. ACM Press, 2009.

[25] J. Kreiker, H. Seidl, and V. Vojdani. Shape analysis of low-level C with overlapping structures. In *VMCAI'10*, volume 5944 of *LNCS*, pages 214–230. Springer, 2010.

[26] O. Lee, H. Yang, and R. Petersen. A divide-and-conquer approach for analysing overlaid data structures. *Formal Methods in System Design*, 41(1):4–24, Apr. 2012. ISSN 0925-9856, 1572-8102.

[27] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear. Verification modulo versions: Towards usable verification. In *PLDI '14*, pages 294–304. ACM Press, 2014.

[28] R. Manevich, T. Lev-Ami, M. Sagiv, G. Ramalingam, and J. Berdine. Heap decomposition for concurrent shape analysis. In *SAS'08*, volume 5079 of *LNCS*, pages 363–377, 2008.

[29] A. Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *ESOP'11*, pages 398–418. Springer, 2011.

[30] A. Miné. Relational thread-modular static value analysis by abstract interpretation. In *VMCAI'14*, volume 8318 of *LNCS*, pages 39–58. Springer, 2014.

[31] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In *ERTS'16*, 2016.

[32] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL'07*, pages 327–338. ACM Press, 2007.

[33] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: An infrastructure for C program analysis and transformation. In *CC'02*, volume 2304 of *LNCS*, pages 213–228. Springer, 2002.

[34] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for detecting races. In *PLDI'06*, pages 320–331. ACM Press, 2006.

[35] P. Pratikakis, J. S. Foster, and M. Hicks. Existential label flow inference via CFL reachability. In *SAS'06*, volume 4134 of *LNCS*, pages 88–106. Springer, 2006.

[36] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Prog. Lang. Syst.*, 27(2):185–235, 2005.

[37] M. D. Schwarz, H. Seidl, V. Vojdani, and K. Apinis. Precise analysis of value-dependent synchronization in priority scheduled programs. In *VMCAI'14*, volume 8318 of *LNCS*, pages 21–38. Springer, 2014.

[38] H. Seidl and V. Vojdani. Region analysis for race detection. In *SAS'09*, volume 5673 of *LNCS*, pages 171–187. Springer, 2009.

[39] H. Seidl, V. Vene, and M. Müller-Olm. Global invariants for analyzing multithreaded applications. *Proc. of the Estonian Academy of Sciences: Phys., Math.*, 52(4):413–436, 2003.

[40] H. Seidl, V. Vojdani, and V. Vene. A smooth combination of linear and Herbrand equalities for polynomial time must-alias analysis. In *FM'09*, volume 5850 of *LNCS*, pages 644–659. Springer, 2009.

[41] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In N. Jones and S. Muchnick, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[42] P. E. Shved, V. S. Mutilin, and M. U. Mandrykin. Experience of improving the blast static verification tool. *Programming and Computer Software*, 38(3):134–142, May 2012.

[43] E. Tomasco, O. Inverso, B. Fischer, S. L. Torre, and G. Parlato. Verifying Concurrent Programs by Memory Unwinding. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 551–565. LNCSS 9035, Springer Berlin Heidelberg, 2015.

[44] V. Vafeiadis. RGSep action inference. In *VMCAI'10*, volume 5944 of *LNCS*, pages 345–361. Springer, 2010.

[45] V. Vojdani and V. Vene. Goblint: Path-sensitive data race analysis. *Annales Univ. Sci. Budapest., Sect. Comp.*, 30:141–155, 2009.

[46] J. W. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *ESEC/FSE'07*, pages 205–214. ACM Press, 2007.