UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Vootele Rõtov

Time Partitioning in Goblint: Extending region analysis with happens-before information

Master's Thesis (30 ECTS)

Supervisor: Vesal Vojdani, PhD
Supervisor: Kalmer Apinis, PhD

Tartu 2016

Time Partitioning in Goblint: Extending region analysis with happens-before information

Abstract: The concurrent nature of device drivers makes them notoriously difficult to manually debug. Goblint, a static analysis framework tries to automatically verify the inexistence of data races. The key challenge in doing that is the precision of the analysis. This paper proposes an enhancement to the region analysis of Goblint to incorporate domain-specific happens-before guarantees. The proposed addition is implemented and evaluated on the Goblint benchmark suite. We show that the given enhancement increases the precision of Goblint when analysing character drivers.

Keywords: Concurrency, race condition, data race, abstract interpretation, happens-before, device drivers, Goblint

CERCS: P170, Computer science, numerical analysis, systems, control

Staatilise analüsaatori Goblinti regioonianalüüsi täiendamine aja partitsioneerimisega ***happens-before*** teabe alusel

Lühikokkuvõte: Seadmedraiverite ehk ohjurite paralleelne olemus muudab nendest vigade leidmise inimese jaoks väga keeruliseks. Staatiline analüsaator Goblint üritab automaatselt verifitseerida, et ohjuris puuduvad andmejooksud. Sealjuures on suureks väljakutseks analüüsi täpsus. Käesolev töö arendab edasi Goblinti regioonianalüüsi, mis võimaldab arvesse võtta valdkonnale eriomaseid *happens-before* tagatisi. Väljapakutud täienduse implementeerimise ning muudatuste mõju analüüsimise aluseks on võetud Linuxi ohjurite alamhulk. Me näitame, et mainitud edasiarendus suurendab Goblinti täpsust *character* tüüpi ohjurite analüüsimisel.

Võtmesõnad: Paralleelsus, andmejooks, abstraktne interpretatsioon, seadmedraiverid, ohjurid, Goblint

CERCS: P170, Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

Device drivers are open programs that work in difficult environment. One of the difficulties is that device drivers have no control over when the callbacks they provide to the Linux kernel are called. This, combined with the fact that device drivers are written in a subset of C, a low level language with very few built-in concurrency abstractions, means that debugging the concurrency issues in device drivers is notoriously difficult. Empirical research confirms that concurrency bugs are common in device drivers [4, 16]. In addition to that, these bugs have a big impact and contribute to a large share of system crashes [24].

Goblint is a sound static analyser for detecting data races in Linux device drivers, developed in the University of Tartu and the Technical University of Munich. Goblint detects a subtype of concurrency issues called data races – situations where more than one thread simultaneously tries to access a shared memory location. If Goblint as a sound tool does not find any data races, it has verified that the driver under analysis is free of data races.The key challenge for Goblint is to assure that the results of analysis would not only be sound, but would also contain as few false-positives as possible. High precision results either in outright verification of the driver or provides the user with valuable feedback. Low precision, however, produces too much noise to be useful. The aim of this thesis is to increase the precision of Goblint by eliminating one type of false-positives. These were found while manually reviewing the data races detected in the Goblint benchmark suite as part of my contribution to a research paper [27].

As part of this thesis, an analysis performed by Goblint was enhanced with time dimension. Inspired by the happens-before relation and motivated by the results of the performed benchmarking, the extra dimension enables Goblint to take into account guarantees provided by the Linux kernel of the way the exposed callbacks are invoked. For example, let there be an assignment to memory location `i` in both of the functions `init` and `exit`. If we know that both functions are called only once and that call to `init` must complete before `exit` can be called then we can safely rule out a data race on `i`. As a result of the added dimension, the analysis of most device drivers in benchmark suite are more precise. For 6 out of 25 drivers in the benchmark suite, the results have improved notably.

The thesis continues with Section 2, where we give an overview of Linux device drivers and also show the extent of impact of bugs on them. Next, in Section 3 we introduce reader to the basics of abstract interpretation, a static analysis technique Goblint uses. Section 4 introduces two key concepts for static analysis of data races. In addition, an overview is given of other notable tools for data race detection in device drivers. An overview of region analysis and theoretical background of my contribution, i.e. time based partitioning, is given in Section 5. Lastly, in Section 6, we give an overview of Goblint, the implementation details of the added enhancement and also the evaluation of effect that the changes had.

2 Device Drivers

To explain the role of device drivers, let's introduce two common devices that are usable on computers running a Linux operating system. The first is `/dev/tty`, a device that represents the terminal controlling the current process [25]. Opening a terminal in Linux and writing to the device results in the text being displayed in the terminal. One can try this with the following command:

`echo "Hello" > /dev/tty`.

The second device we will introduce is `/dev/null`, a device that discards any data sent to it. So, redirecting text to this device has no effect. One can send text to the device as follows:

`echo "Hello" > /dev/null`.

Although the two devices act differently, they can be both used in the same way – we can redirect text to them (and redirect text from them). This allows us to group the two devices together when reasoning about how to use them. The benefits of this are far greater when the number of devices that can be grouped together grows.

The role of Linux device drivers is to enable such groupings that can contain an arbitrary amount of devices. This is done by enabling access to the devices via a small set of well-defined interfaces that other parts of the Linux kernel can make use of.

In this paper, we will focus on a subset of device drivers called *character (char) device drivers*. Character device drivers allow unbuffered access to the underlying "hardware". Both of the drivers for the example devices above are char drivers. We will first consider the common interface used to access the operations of the device drivers and then discuss what can go wrong when these drivers execute in a concurrent setting.

2.1 Interface of a Character Device Driver

Devices are exposed to the end user as entries in the file system and therefore it is natural to use the same terminology for talking about device drivers' operations as about normal file operations. Device drivers expose endpoints for *reading*, *writing to*, *opening* and *releasing* devices. This list is not exhaustive[25, include/linux/fs.h], but it is not necessary for a driver to expose them all.

As a running example we will introduce the driver `Counter`. Counter keeps track of the difference between how many times the kernel has read from it and how many times the kernel has written to it [1].

In the following code snippet, the parameters of the functions are omitted as they are not relevant for this example.

```
static ssize_t file_read(...){
    ++i;
    printk("Increasing i, new value:: %d \n",i);
    ...
}

static ssize_t file_write(...){
    --i;
    printk("Decreasing i, new value: %d \n",i);
    ...
}

static int file_open(...)
{
    printk("Opening device \n");
    return 0;
}

static int file_release(...){
    printk("Closing device \n");
    return 0;
}
```

The `read` function increases the global counter by one, while the `write` function decreases it by one. Both the `open` and `release` functions only output info to the kernel log.

When the user reads from a file that exposes the driver, one can see that the file was opened, read and then closed. For a concrete example, `head -c 1 < counter` produces the following entries in the kernel log:

```
vootele kernel: Opening file
vootele kernel: Increasing i, new value: 1
vootele kernel: Closing file
```

And similarly, when writing to the file, for example with `echo hi > my-driver`, the file is opened, written into and then closed.

[1]The full source code is available in Appendix A

Drivers expose devices to the kernel via `file operations` structure. When registering a driver to the kernel, the file operations structure is registered with the kernel, which can invoke any of the methods specified in the structure.

The file operations structure of Counter is

```
static struct file_operations f_ops = {
  .write = file_write,
  .read = file_read,
  .open = file_open,
  .release = file_close,
};
```

Device drivers also expose `init` and `exit`, which the kernel can use to register and de-register a device driver. In case of Counter, registering the device initializes the counter `i` to zero and then registers the file operations.

2.2 Data Races in Device Drivers

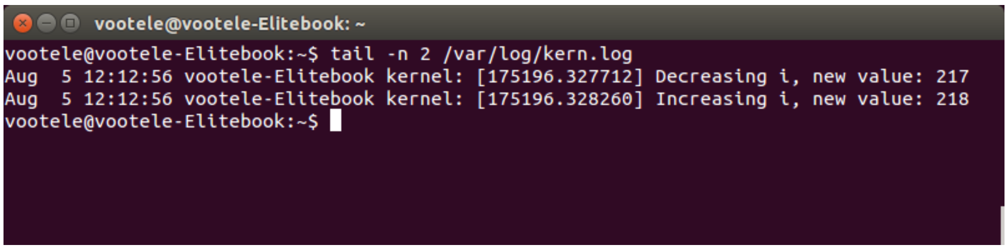
When registering our driver, we exposed multiple endpoints to the kernel. From that moment onwards, kernel can invoke any of the registered methods, at any time, until the driver is de-registered.

The multitude of entry points to the programs and lack of control over when they are entered makes avoiding *data races* a challenge when writing Linux device drivers. We say that data race occurs in a program when different threads simultaneously attempt to access a shared memory location and at least one of the accesses is a write operation.

For an example of a data race in device drivers, it is quite obvious that when one writes to a device exposed by the example driver concurrently, then the value of the counter after all the write operations is not determined – it is easily possible that between the read and write operations of the counter, its value gets updated.

To be more concrete, in Figure 1 is a snippet of the kernel log after running a script [2] that reads and writes to the example driver in a loop that runs 10 000 times in 3 processes at the same time. As the read operation increases i and the write operation decreases it, the expected value without a data race would have been 0.

[2] Available in Appendix B



```
vootele@vootele-Elitebook:~$ tail -n 2 /var/log/kern.log
Aug  5 12:12:56 vootele-Elitebook kernel: [175196.327712] Decreasing i, new value: 217
Aug  5 12:12:56 vootele-Elitebook kernel: [175196.328260] Increasing i, new value: 218
vootele@vootele-Elitebook:~$
```

Figure 1: Snippet from the kernel log after data race in an example device driver.

To avoid data races, Linux kernel offers multiple locking primitives.

A popular locking primitive is the *spinlock*, which runs a tight loop, inside which it tries to acquire the lock until it succeeds. This tight loop gives the lock its name. Using spinlocks avoids explicitly putting the thread to sleep when acquiring a lock fails. If it is known that lock is held only for a very short time then the computational cost of putting a thread to sleep and then later awakening it outweights the extra CPU cycles used by the tight loop of a spinlock.

Spinlocks also offer a lock that differentiates between reads and writes, allowing multiple concurrent reads.

One of the most commonly used locking primitives is the *mutually exclusive lock* (mutex). It also the one that should be used, if possible [25, locking/mutex-design.txt]. Older implementations of mutex put the thread to sleep if acquiring the lock failed, whereas the current implementation is also capable of spinning for a limited time before resorting to explicitly putting the thread to sleep.

To eliminate a possible data race from our example driver, we could add a lock to the driver and acquire it on every read and write operation. With the added lock, the read operation would look as follows:

```
static ssize_t file_read(...){
    mutex_lock(&my_mutex);
    ++i;
    printk("Increasing i, new value: %d \n",i);
    mutex_unlock(&my_mutex);
    ...
}
```

Here, `my_mutex` is a static mutex.

After adding the same locking pattern to write operations, data race will no longer plague the example driver.

2.3 Impact of Data Races on Device Drivers

We have seen that data races are a real possibility in device drivers and also that there are ways to protect oneself against them.

As previously discussed, Linux device drivers usually have multiple entry points and no control over when they are entered. Also, the device drivers are written in C, a low level language, that makes reasoning about them quite challenging. Furthermore, quite often the person writing a driver is an expert on the device that the driver is for, and not so much an expert on the Linux device drivers themselves. This makes correct usage of constructs that help to avoid data races difficult and error-prone.

Empirical studies validate this assertion. In a study done by Chou *et al.* [4] it was found that error rates in device drivers are three to seven times higher than in rest of the kernel.

The situation seems to have become better over time. In the follow-up study conducted by Palix *et al.* [16], the error rate in device drivers improved, but drivers still contained the highest number of errors.

In a study done by Ryzhyk and others [19], out of the 498 bugs found between 2002 and 2008 in 13 selected Linux device drivers, 93 were concurrency related – mainly data races or deadlocks.

Mutilin *et al.* [12] found that data races are the most common single type of bug, remarkable 5 times more common than deadlocks in the Linux device drivers.

It is worth noting that based on [4, 16], the average lifetime of a bug is 18 months, making it quite likely that the bug makes its way to a stable version.

In addition to being common, the bugs in device drivers can cause crashes fatal to the whole system. In Windows XP, 85% of the reported failures were caused by issues with the device drivers [24].

The data races are extremely unpleasant in safety-critical applications, where their presence could endanger lives. For that reason, the verification requirements of device drivers in safety-critical domains (for example aviation or automobile industry) are very rigorous, making the testing process very costly and time-consuming.

3 Theoretical Foundations of Static Analysis

By *program analysis* (or just *analysis*) we mean a process of deciding whether some property holds for a program under analysis. Analysis of a program is *static* if the program being analysed does not get executed during the analysis, in contrast to *dynamic* analysis, during which program is executed. Static analyses are often used by compilers, for example for finding uses of variables that are not declared beforehand in Java or C programs. In addition, modern IDEs constantly run wide selection of static analyses in the background, to enable things such as variable renaming, for which one would have to know about all the usages of a specific variable in the program, taking into account the relevant scopes.

There are good reasons for preferring a static analysis over a dynamic (or combined) one. To pick a few, it allows us to say something about programs that do not compile (and as such, cannot be analysed dynamically) and also about open programs, such as modules and device drivers, that cannot be run in isolation, but can, however, be analysed statically. It is faster than a dynamic analysis, which cannot be performed faster than the time it takes to execute the program. As static analyses do not depend on any information gathered from a specific run, it is possible to have an analysis that says something about all the possible runs of a program under all possible behaviors of the environment (and not only, say, for all the possible runs that the analysis has witnessed).

As one might guess, there are a lot of techniques that fit the wide definition given for the static analysis. We will be focusing on one of them — *abstract interpretation* [5]. Unlike most static checkers found in IDEs, abstract interpretation does not simply search for bug patterns; instead, it attempts to compute all possible behaviors of the system in a way that is mathematically reliable.

To give intuition into what is an abstract interpretation, consider the following program:

```
int main(void) {
  int x, y, z;
  x = 0;
  y = -81;
  z = -37
  if (y * z * rand()> 0){
    x = 1;
  }
  return x;
}
```

One can easily deduce that the program will always exit with the error code 1 [3], without having to calculate the exact value of `y * z * rand()`, which would not be possible statically. Instead, we can interpret all the values the variables have as negative ($-$), 0 or positive ($+$) and evaluate `y * z * rand()` as $- * - * + = +$.

With this simplification, we do lose precision. Let's consider another example:

```
int main(void) {
  int x, y, z;
  x = 0;
  y = -81;
  z = -37;

  if (y * z > 0){
    x = 1;
  }

  if (y * z -y > 0){
    x = 2;
  }
  return x;
}
```

Here, using the interpretation that we applied successfully in the previous example, we would not be able to decide on the return value of the program. When evaluating `y * z - y`, we can rule out any of the possible values. To be able to evaluate the product in a similar way to previously described, we could assign the variables a set of *possible* values instead. In this case, we would evaluate `y * z - y` as $\{-\} * \{-\} - \{-\} = \{-, 0, +\}$. Although we cannot decide what the program will return, we can decide based on the analysis that the first `if` block can safely be ignored – an optimization a compiler could perform.

In the following, the definitions have been inspired by [26, 2].

3.1 Concrete Semantics

To build a mathematically sturdy foundation for abstract interpretation we first need to give a more formal definition for executing a program. This is known as program's *concrete semantics*. To be able to formalize concrete semantics, we first need to define what we mean by a program. In this section we will presume

[3]Presuming that the constant `RAND_MAX` has not been redefined.

that programs are represented as flow graphs. The flow graphs serve as a common intermediate representation that is powerful enough to express all programs written in higher level languages and at same time, considerably easier to mathematically reason about.

Definition 3.1. *Procedure p is a control flow graph (N_p, E_p, s_p, r_p), where N_p is a finite set of nodes, $s_p \in N_p$ is the entry node with in-degree 0 and $r_p \in N_p$ is the return node with out-degree 0. Set of labels L consists of*

- *statements $s \in Stmt$, other than procedure calls,*
- *procedure calls $f()$, where $f \in Exp$ is an expression,*
- *positive and negative guards ($Pos(e) \in Guards$ and $Neg(e) \in Guards$, $e \in Exp$),*
- *and thread spawning function $spawn(e), e \in Exp$.*

The edge set $E_p \subseteq N_p \times L \times N_p$. In addition, we require that out-degree of node $n \in N_p$ is no bigger than 2 and if e_1 and e_2 are outgoing edges of n and $e_1 \neq e_2$, then one of e_1 and e_2 is a positive guard and other a negative guard.

It is worth mentioning that although we have constrained ourselves with procedure calls without any arguments and only one return node, those issues can be easily solved by using global variables.

In addition, for simplicity, we assume that for every $n \in N_p$, there exists a path from s_p to n and from n to r_p.

Definition 3.2. *Program $P = (Proc, main)$, where $Proc$ is a finite set of procedures such that for every $p, q \in Proc, p \neq q \implies N_p \cap N_q = \emptyset$ and $main \in Proc$ is one of the procedures designated as the main function. Let N be set of all nodes in the program, that is $N = \bigcup_{p \in Proc} N_p$ and E be a set of all edges in the program, that is $E = \bigcup_{p \in Proc} E_p$.*

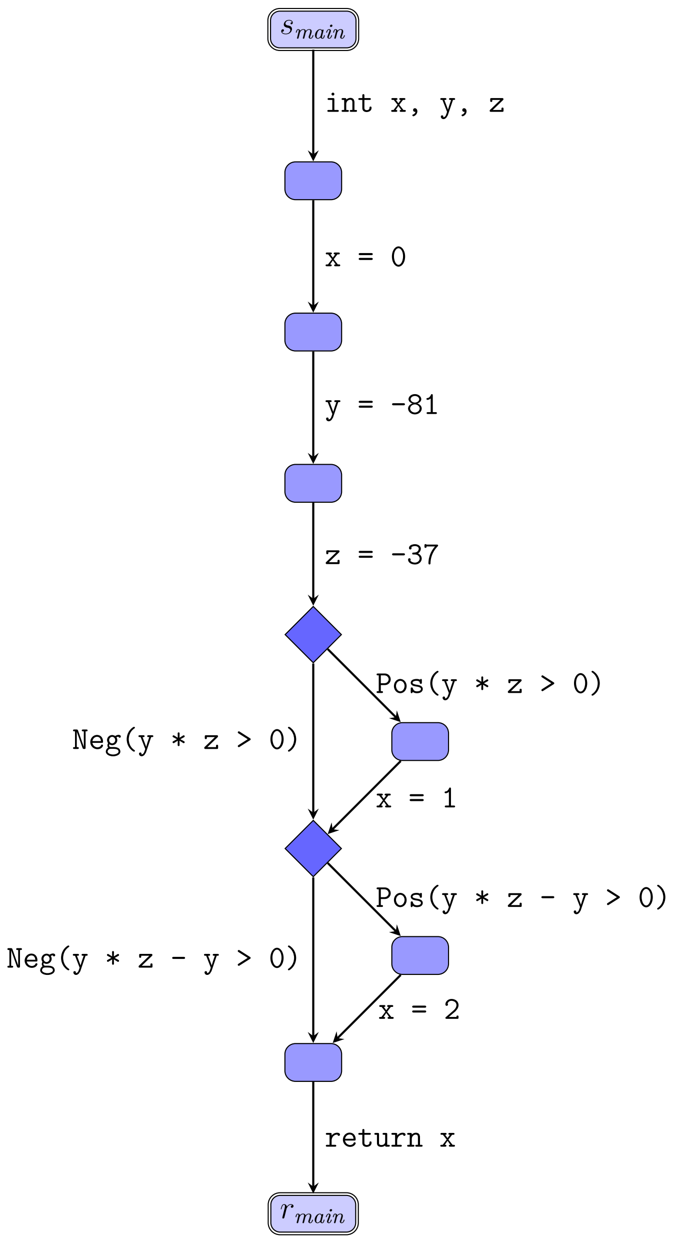


Figure 2: CFG corresponding to the last code snippet.

As it is not feasible to offer a formalization of all the statements and expressions available in any common higher-level programming language as a part of this thesis, we have elected to leave the sets *Stmt* and *Exp* formally undefined, instead relying on the reader's previous experience with statements and expressions in any of the C family languages. For the interested, a formalization for a subset of C can be found in [26, 17].

Now that we have defined how a program looks like, we will continue with how such a program is evaluated – the *semantics* of the program.

From here on we will use the following notation to "update" a function:

$$f\left[x:a\right]\left(y\right)=\begin{cases}a & \mathit{if}\ y=x\\ f\left(y\right) & \mathit{otherwise}\end{cases}$$

.

Let's first consider single-threaded programs. Let S be a set of states of the program, that is $s : \mathit{Var} \to \mathit{Val}$, where Var is a set of variables in the program and Val is a set of possible values. Again, we will not formally define sets Val and Var here. For every statement $\mathit{stmt} \in \mathit{Stmt}$, let $[\![\mathit{stmt}]\!]_{\mathit{Stmt}} : S \to S$ be a function that updates the state of the program.

For example

$$[\![\texttt{x = 3}]\!]_{\mathit{Stmt}}\left(s\right)=s\left[x:3\right].$$

Similarly, for every expression $e \in \mathit{Exp}$, let $[\![e]\!]_{\mathit{Exp}} : S \to \mathit{Val}$ that evaluates expression in the context of the state.

For example, let s be a state after assigning 3 to x, that is

$$s=[\![\texttt{x = 3}]\!]_{\mathit{Stmt}}\left(s_0\right),$$

where s_0 is an arbitrary state.

Then

$$[\![\texttt{x + 8}]\!]_{\mathit{Exp}}\left(s\right)=11.$$

With this in mind, we can define a relation for intra-procedural evaluation of a procedure as follows:

Definition 3.3. *Intra-procedural evaluation relation of procedure p, $\hookrightarrow_p$, is a relation between $N_p \times S$ and $N_p \times S$ for which following rules hold:*

$$\textit{Stmt}\frac{(u, stmt, v) \in E_p \qquad [\![stmt]\!]_{Stmt}(s) = s'}{(u, s) \hookrightarrow_p (v, s')}$$

$$\textit{Pos}\frac{(u, Pos(e), v) \in E_p \qquad [\![e]\!]_{Exp}(s) = true}{(u, s) \hookrightarrow_p (v, s)}$$

$$\textit{Neg}\frac{(u, Neg(e), v) \in E_p \qquad [\![e]\!]_{Exp}(s) \neq true}{(u, s) \hookrightarrow_p (v, s)}.$$

This relation gives a formal definition for one atomic step during evaluation of a procedure, when evaluating a procedure step with the relation $\hookrightarrow$, the rule applied depends on the type of the edge under evaluation.

To support inter-procedural evaluation of our program, we need to keep track of the caller of the process. For that we will use, as it is usually done, a *call stack*. At every function call, we add a node where the call was made from to the stack. When a return node is reached, a node will be popped from the stack and the program evaluation will continue from there. More formally, a call stack is a tuple of CFG nodes. Let *Stack* be a set of all call stacks of the program P.

We will define inter-procedural evaluation relation for single-threaded program P as follows:
Definition 3.4. *Inter-procedural evaluation relation of single-threaded program* $P = (Proc, p_{main})$*,* $\dashrightarrow$*, is a relation between* $Stack \times S$ *and* $Stack \times S$ *for which following rules hold:*

$$\textit{IntraProc}\frac{(u, l, v) \in E \qquad \exists p \in Proc\,(u, s) \hookrightarrow_p (v, s')}{(u :: xs, s) \dashrightarrow (v :: xs, s`)}$$

$$\textit{Call}\frac{(u, f(), v) \in E \qquad [\![f]\!]_{Exp}(s) = p \qquad p \in Proc}{(u :: xs, s) \dashrightarrow (s_p :: v :: xs, enter_p s)}$$

$$\textit{Return}\frac{p \in Proc}{(r_p :: xs, s) \dashrightarrow (xs, return_p s)}$$

where enter$_{proc}$ and return$_{proc}$ are state transformers that initialize and destroy local variables.

To support multithreaded programs, we will first expand our inter-procedural evaluation relation to support spawning of other threads. We will later use this to define semantics that takes thread interleavings into account.

Definition 3.5. *Intra-thread evaluation relation of P,$\twoheadrightarrow$, is a relation between ($Stack \times S$) and $Stack \times S \times Proc^*$ for which following rules hold:*

$$\textit{Spawn}\frac{(u, \textit{spawn}(f), v) \in E \qquad [\![f]\!]_{Exp} = p \qquad p \in \textit{Proc}}{(u :: xs, s) \twoheadrightarrow (v :: xs, s, [p])}$$

$$\textit{IntraThread}\frac{(u, l, v) \in E \qquad (xs, s) \dashrightarrow (ys, s')}{(xs, s) \twoheadrightarrow (ys, s', [])}.$$

The added information about spawned threads will be used by the following relation.

Definition 3.6. *The inter-thread evaluation relation of P, $\Rrightarrow$, is a relation between $Stack^* \times S$ and $Stack^* \times S$ for which following rule holds:*

$$\frac{0 \leq i \leq n \qquad (t_i, s) \twoheadrightarrow (t'_i, s', [p_1, \ldots, p_k])}{((t_0, \ldots, t_i, \ldots, t_n), s) \Rrightarrow ((t_0, \ldots, t'_i, \ldots, t_n, [s_{p_1}], \ldots, [s_{p_k}]), s')}.$$

In the ($\Rrightarrow$) relation, the list of stacks corresponds to call stacks of all the threads currently running. It is worth noting that relation ($\Rrightarrow$) is non-deterministic if there is more than one thread in the program – a property that multithreaded programs have.

The ($\Rrightarrow$) lets us define a set of all possible states of the program P, with starting state s_0 :

$$\mathcal{S} = \{z | ((s_{main}), s_0) \Rrightarrow^* z\}.$$

Equipped with set $\mathcal{S}$, we would have information about the program at any possible execution point. At the same time, it is clear that computing the set $\mathcal{S}$ is not feasible in most cases — the set might not be finite and even if it is, all the possible states of even a simple program might be quite big.

3.2 Abstract Domains

As in the example we gave at the start of this section, we can, however, simplify the program by *abstracting* away parts of the state that are not interesting for us for finding out if a certain property holds for the program under analysis. However, it is not easy to get the level of abstraction right, as abstracting away too much does not let us tell much about non-trivial properties of the program and abstracting away too little leaves us with a task that is too big to feasibly compute.

To have our abstraction on a sound footing, we need a formalization of the idea. The mathematical theory of *complete lattices* offers a suitable framework. We will now give a short introduction to the lattice theory.

First of all, a quick reminder from the set theory.

Definition 3.7 (Partial Order). *A set D with relation $\sqsubseteq$ on D is called a **partially ordered set** if $\sqsubseteq$ is a **partial order**, that is reflexive, antisymmetric and transitive.*

Definition 3.8 (Upper bound). *Let $(D, \sqsubseteq)$ be a partially ordered set. An **upper bound** of set $X \subseteq D$, is an element $x \in D$, such that for every element in $y \in X$, $y \sqsubseteq x$. Element x is the **least upper bound** of set X, denoted as $\bigsqcup X$ if for every other upper bound z of X, $x \sqsubseteq z$.*

Definition 3.9 (Lower bound). *Let $(D, \sqsubseteq)$ be a partially ordered set. An **lower bound** of set $X \subseteq D$ is an element $x \in D$, such that for every element in $y \in X$, $x \sqsubseteq y$. Element x is the **greatest lower bound** of set X, denoted as $\bigsqcap X$ if for every other lower bound z of X, $z \sqsubseteq x$.*

Equipped with these three definitions, we can now define *complete lattice*:

Definition 3.10 (Complete lattice). *A tuple $(D, \sqsubseteq)$ is a complete lattice if D is a partially ordered set with relation $\sqsubseteq$ and for every set $X \subseteq D$, there exists $\bigsqcup X$ and $\bigsqcap X$.*

We will use notation $x \sqcup y$ to denote $\bigsqcup \{x, y\}$ and analogously, $x \sqcap y$ to denote $\bigsqcap \{x, y\}$. In addition, we will use $\perp$ to denote the least element in D, that is $\perp = \bigsqcap D$ and $\top$ to denote the greatest element in D, $\top = \bigsqcup D$.

Lets now have a look at couple of examples.

First of all, let D be $\{\mathit{false}, \mathit{true}\}$ with the ordering $\mathit{false} \sqsubseteq \mathit{true}$. Then $\mathit{true} = \top = \bigsqcup D$ and $\mathit{false} = \perp = \bigsqcap D$. A usual way to describe lattices is the use of Hasse diagrams, graphs that have as nodes the elements of D and there is an edge between $u, v \in D$ if $u \sqsubseteq v$. The edges that are implied by reflexivity or transitivity are usually omitted.

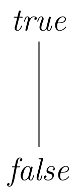


Figure 3: Lattice D as an Hasse diagram.

Secondly, in the example we used to gain intuition into abstract interpretation, we gave each variable a set of its possible values at that program state. For the values we differentiated between negative integers, zero and positive integers. The corresponding lattice would be the following.

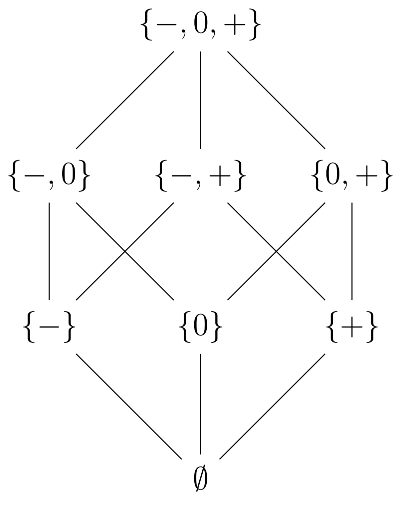


Figure 4: Lattice of subsets of $\{-,0,+\}$, ordered by inclusion.

It is worth noting that for every set S, its powerset 2^S is a complete lattice when ordered by inclusion with union of two sets being the least upper bound and intersection being the greatest lower bound.

As a last example, let's look at *flat lattice* – a lattice defined on a set $S \cup \{\bot, \top\}$, with following ordering $\sqsubseteq = \{(u, v) \,|\, u = \bot \wedge v \in S \vee u \in S \wedge v = \top\}$. For a concrete example, let S be $\mathbb{Z}$. Then the lattice would look as follows.

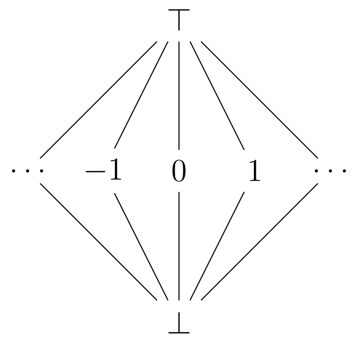


Figure 5: The flat lattice of $\mathbb{Z}$.

The flat lattice is one way to create a complete lattice from any kind of set.

Lattices will serve as *abstract domains* for abstract interpretation. Intuitively, the elements of abstract domain will fulfill the same role as the states in concrete interpretation, offering context for interpreting the program. For abstract domain (that is, a complete lattice) $D, \sqsubseteq$, we will choose D to be a set of elements that each describe some set of states of our concrete implementation. To analyse the program we would like to find for every point $n \in N$ in our program the most precise element $d \in D$ that describes the program state at that point. To expand, we want d to be such that it describes the least amount of states in our concrete implementation, while still describing all the states that our program could be at point n. So, an *analysis* of program P is a function $\mathcal{A} : N \to D$.

For our running example, as we only care about the signs of variables, a suitable D would be a set of functions from the set of all variables in the program P to the power set of $\{-, 0, +\}$, that is $d \in D, d : \mathit{Var}_P \to 2^{\{-,0,+\}}$. D is ordered point-wise, that is $d_1 \sqsubseteq d_2 \iff \forall v \in \mathit{Var}_p, d_1 v \subseteq d_2 v$.

It is worth noting that D is also a lattice as for every subset X,

$$\bigsqcup X = \left\{ (v, b) \,| \forall v \in \mathit{Var}, b = \bigsqcup \{ d\,(v) \,| d \in X \} \right\}$$

and $\bigsqcap X$ is defined dually.

One might still wonder why do we want to use lattices here, instead of simply using sets without the partial order or the lattice operations as abstract domains. Consider the following program.

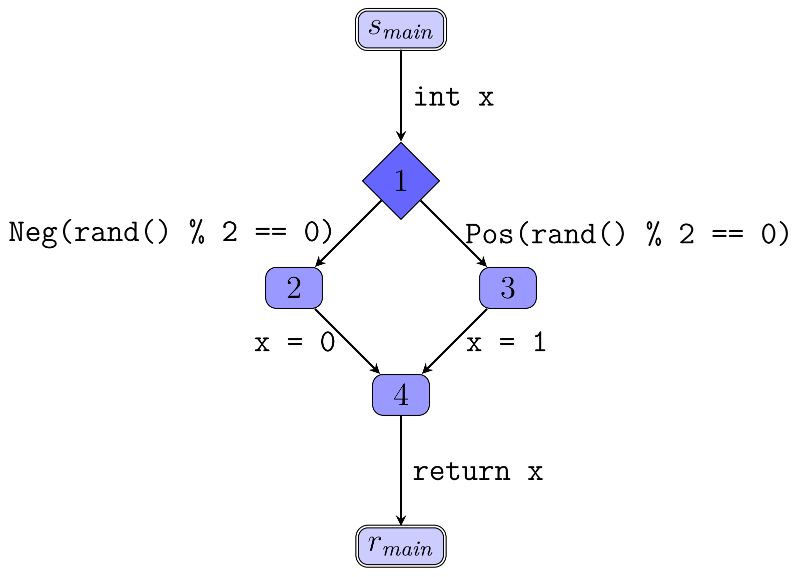


Figure 6: An example program.

When taking the *Pos* edge in the example program then the most precise element of the domain that would describe the state at node 4 would be $d_1 = \{(x, \{+\})\}$. When taking the *Neg* edge, the most precise element for describing the state at node 4 would be $d_2 = \{(x, \{0\})\}$. However, based on static analysis, it is not possible to say which edge will be picked as it depends on a random number generated at runtime. In this case, we would like to describe the state at node 4 with an element from domain that describes the state most precisely, taking into account that either of the conditional branches could be taken. Based on how we have constructed this domain, the suitable element would be $d_1 \bigsqcup d_2 = \{((x, \{0, +\})\}$.

To be able to use the least upper bound when joining information from different possible incoming edges to state, the domain should be ordered in a way that if $d_1 \sqsubseteq d_2$ then if a property P that we are interested in holds for d_1, it should always hold for d_2 as well. More concisely, $d_1 \sqsubseteq d_2 \implies (P(d_1) \implies P(d_2))$.

If the domain is suitably ordered, then the lattice operations provide us with a very convenient and elegant way to approximate the state of the program as precisely as possible while also providing us with an interface that lets us abstract away the details of how the lattice operations are defined when reasoning about abstract domains in general.

In our example, the properties we were considering were whether a specific variable's value *may* be positive, negative or zero. If the value of a variable may be

0, then it also may be 0 or positive, so we would want $\{0\} \sqsubseteq \{0,+\}$. However, if we would wonder about whether a specific variable's value would *not* be positive, negative or zero then if we know that the value is not zero or positive then it is also not zero, meaning that we would want to have a lattice $\left(2^{\{-,0,+\}}, \sqsubseteq_{must}\right)$ such that $\{0,+\} \sqsubseteq_{must} \{0\}$. We could define $\sqsubseteq_{must}$ to be

$$d_1 \sqsubseteq_{must} d_2 \iff d_2 \sqsubseteq d_1.$$

and the least upper bound to be intersection of two sets and greatest lower bound to be union of the sets.

More formally, let P be a program, $\mathcal{S}$ be set of all the possible states of the program and let $L = (D, \sqsubseteq)$ be a lattice. We say that $\Delta \subseteq \mathcal{S} \times D$ is a *descriptor relation* if there is no $s \in \mathcal{S}$ such that $s\Delta\bot_L$, for every state $s \in \mathcal{S}$, $s\Delta\top_L$ and for all $d_1, d_2 \in D, s \in \mathcal{S}$,

$$s\Delta d_1 \wedge d_1 \sqsubseteq d_2 \implies s\Delta d_2.$$

In concrete interpretation, to evaluate a program statement (v, s, u) we used function $[\![s]\!]_{Stmt} : S \to S$. To evaluate the same program statement in abstract interpretation over domain D, we need to define functions $[\![s]\!]^*_{Stmt} : D \to D$, that we can use to evaluate the program in the context of domain element d. For example, in case of our running example, we could define $[\![\texttt{x = x + 2}]\!]^*_{Stmt}$ as

$$[\![\texttt{x = x + 1}]\!]^*_{Stmt}(d) = \begin{cases} d\,[x : \{+\}] & d\,(x) \in \{\{0\}, \{+\}, \{0,+\}\} \\ d\,[x : \{-,0\}] & d\,(x) = \{-\} \\ d\,[x : \{-,0,+\}] & d\,(x) \in \{\{-,0\}, \{-,+\}, \{-,0,+\}\} \end{cases}.$$

We demand that the functions $[\![s]\!]^*_{Stmt}$ are consistent with functions $[\![s]\!]_{Stmt}$, that is, for each $s \in S, d \in D$ if $s\Delta d$ then $[\![s]\!]_{Stmt}\,\Delta\,[\![d]\!]^*_{Stmt}$. This relation is illustrated in the following figure.

$$\begin{tikzcd}
s = \{(x,0)\} \arrow[r, dash, dashed, "s\Delta d"] \arrow[d, "s' = [\![\texttt{x = x +1}]\!]_{Stmt}(s)"'] & d = \{(x,\{0\})\} \arrow[d, "d' = [\![\texttt{x = x + 1}]\!]^*_{Stmt}(d)"] \\
s' = \{(x,1)\} \arrow[r, dash, dashed, "s'\Delta d'"] & d' = \{(x,\{+\})\}
\end{tikzcd}$$

Figure 7: The connections between $\mathcal{S}$, D, $[\![\texttt{x = x +1}]\!]_{Stmt}$ and $[\![\texttt{x = x + 1}]\!]^*_{Stmt}$.

Also, we will want all the functions $[\![s]\!]^*_{Stmt}$ to be *monotonic.*
Definition 3.11. *Let $(X, \leq_X)$ and $Y, \leq_Y$ be partially ordered sets. Then $f : X \to Y$ is monotonic if for every x_1, x_2, $x_1 \leq_X x_2 \implies f(x_1) \leq_y f(x_2)$.*

This requirement is quite natural – the monotonicity of the evaluation functions means that if we abstractly evaluate a program statement in the context of domain element, then the resulting state has to be at least as precise as if we would have evaluated the same statement in the context of less precise domain element.

3.3 Abstract Interpretation

In this subsection, let's limit ourselves to intraprocedural analyses for simplicity.

As mentioned before, we want to define the analysis $\mathcal{A} : N \to D$ such that for every node n in program P, every state $s \in \mathcal{S}$ that can happen when the execution has reached node n, $s\Delta f(n)$.

Let n be a node in program P. Let w be a finite walk in program P, starting from state s_P and ending in n, that is $w = (s_P, stmt_0, s_1, stmt_1, \ldots, s_i, stmt_i, n)$. Let's note that if d_s is the abstract starting state, that is $s_P\Delta d_s$, then we can evaluate the edges on this path in the following way:

$$d_w = [\![stmt_i]\!]^*_{Stmt} \circ [\![stmt_{i-1}]\!]^*_{Stmt} \circ \ldots \circ [\![stmt_0]\!]^*_{Stmt}(d_s)$$

If we define W_n to be the set of all the finite paths from s_P to n in program P, then could now define f as

$$\mathcal{A}\left(n\right) = \bigsqcup \left\{d_w \| p \in W_n\right\}.$$

This approach, known as *meet over all paths (MOP)*, has downsides, the most obvious is the possible computational complexity as the number of paths can grow exponentially with the size of the program and, even worse, the number of walks do not have to be finite. Additionally, it is not clear how to handle infinite walks.

We will be using another approach for defining $\mathcal{A}$. Instead of considering all paths and joining information at the very end, we will join information as soon different paths converge. It can be shown that this solution over-approximates the MOP solution [15, 80].

Let's consider a specific edge in our program, (v, s, u). Now, let $d = \mathcal{A}\left(v\right)$. Then, presuming that d is precise, $\mathcal{A}\left(u\right)$ should not be more precise than $[\![s]\!]_{Stmt}\left(d\right)$ as otherwise we would have gained more information about the program using our abstract transfer function than when using the concrete transfer function.

From this, we get the following constraint system

$$\mathcal{A}\left(u\right) \sqsupseteq [\![s]\!]_{Stmt}\left(\mathcal{A}\left(v\right)\right) \quad \forall\left(e = (u, s, v)\right) \in E_P. \tag{1}$$

Additionally, the abstract starting state should also be a lower bound to $\mathcal{A}\left(s_P\right)$, so $\mathcal{A}\left(s_p\right) \sqsupseteq d_S$.

We can now define $\mathcal{A}$ to be a function that is the least solution to the given constraint system.

As an example, let's look at the example program B:

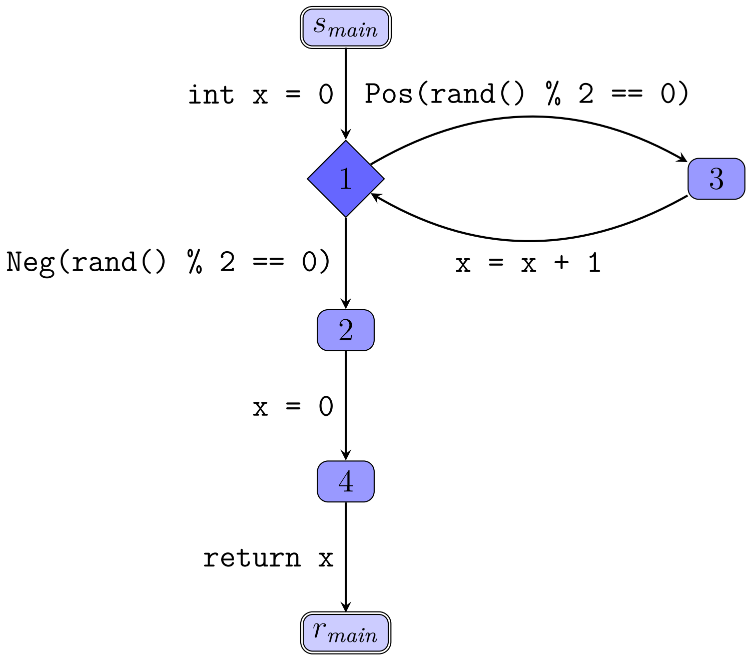


Figure 8: An example program B.

Let $d_s = \{(x, \{\})\}$ be the abstract starting state. Then

$$\mathcal{A}_B : N_B \to \left(\mathit{Var} \to 2^{\{-,0,+\}} \right)$$

must satisfiy the following constraint system

$$\begin{aligned}
\mathcal{A}_B\left(s_{main}\right) &\sqsupseteq d_s \\
\mathcal{A}_B\left(1\right) &\sqsupseteq [\![\texttt{int x = 0}]\!]^*_{\mathit{Stmt}}\left(\mathcal{A}_B\left(s_{main}\right)\right) \\
\mathcal{A}_B\left(1\right) &\sqsupseteq [\![\texttt{x = x +1}]\!]^*_{\mathit{Stmt}}\left(\mathcal{A}_B\left(3\right)\right) \\
\mathcal{A}_B\left(2\right) &\sqsupseteq [\![\texttt{Neg(rand() \% 2 == 0)}]\!]^*_{\mathit{Stmt}}\left(\mathcal{A}_B\left(1\right)\right) \\
\mathcal{A}_B\left(3\right) &\sqsupseteq [\![\texttt{Pos(rand() \% 2 == 0)}]\!]^*_{\mathit{Stmt}}\left(\mathcal{A}_B\left(1\right)\right) \\
\mathcal{A}_B\left(4\right) &\sqsupseteq [\![\texttt{ x = 0 }]\!]^*_{\mathit{Stmt}}\left(\mathcal{A}_B\left(2\right)\right) \\
\mathcal{A}_B\left(r_{main}\right) &\sqsupseteq [\![\texttt{return x}]\!]^*_{\mathit{Stmt}}\left(\mathcal{A}_B\left(4\right)\right).
\end{aligned}$$

It is clear that the constraint system does have a solution – if we take $\mathcal{A}_B\left(x\right) = \top$, then all the constraints will be satisfied. However, the result is of course very imprecise. To achieve as precise result as possible we would like to find the *least* solution to this constraint system.

In the following, we will see when can we find the least solution to the constraint system and how to find such an solution. For this, let's give a more formal definition for constraint system.

Definition 3.12. *A constraint system C over lattice $(D, \sqsubseteq)$ is a set of constraints – pairs (v, f), where v is a constraint variable $v \in Var_C$, and f is a function from assignment of constraint variables $Var_C \to D$ to domain element D.*

A solution is such an assignment of variables $s : Var_C \to D$ that for every constraint (v, f) , $s(v) \sqsupseteq f(s)$.

A classic result from the lattice theory will help us in determining when we can find a least solution to the constraint system.

Definition 3.13. *A strictly ascending chain of size k in lattice $(D, \sqsubseteq)$ is a tuple $(d_1, \ldots, d_k)$, $d_i \in D$, such that*

$$\bot \sqsubset d_1 \sqsubset d_2 \sqsubset \ldots \sqsubset d_k.$$

Definition 3.14 (Lattice height)**.** *The height of lattice $L = (D, \sqsubseteq)$ is h if it is the cardinality of the largest strictly ascending chain.*

Definition 3.15. *An element $x \in X$ is a fixed-point of function $f : X \to X$ if $f(x) = x$.*

Definition 3.16. *Let $L = (D, \sqsubseteq)$ be a lattice and C a chain in that lattice, that is a tuple $(d_1, d_2, \ldots, d_k, \ldots)$ such that*

$$\bot \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots \sqsubset d_k \ldots.$$

We say that the chain stabilizes at index i if for every d_j where $i \leq j$, $d_j = d_i$.

Theorem 3.1 (Kleene's fixed point iteration)**.** *Let $L = (D, \sqsubseteq)$ be a lattice and $f : D \to D$ be a monotonic function. Then if chain*

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots \sqsubseteq f^n(\bot) \sqsubseteq \ldots$$

stabilizes at index i, then f^i will be the least fixed point of f. Furthermore, if the height of L is finite, then the chain will stabilize.

Let C be a constraint system with constraint variables $Var_C = \{v_1, v_2, \ldots, v_n\}$ over lattice $L = (D, \sqsubseteq)$. Then we can find a solution to this constraint system by solving the following inequation

$$\vec{d} \gg F\left(\vec{d}\right) \tag{2}$$

where $\vec{d} : D^n$, $F : D^n \to D^n$

$$F\left(\vec{d}\right) = \left(f_1\left(\vec{d}\right), \ldots, f_n\left(\vec{d}\right)\right),$$

$$h_i\left(d_1, \ldots, d_n\right) = \bigsqcup_{(v_i, g)} g\left(\lambda v_i . d_i\right)$$

and $\gg$ is defined point-wise, that is

$$\left(d_1, \ldots, d_n\right) \gg \left(d_1', \ldots, d_n'\right) \iff d_1 \sqsupseteq d_1' \wedge d_2 \sqsupseteq d_2' \wedge \ldots \wedge d_n \sqsupseteq d_n'.$$

It is easy to verify that if L is a lattice, then also $M = \left(D^n, \gg\right)$ is a lattice. Let $\left(d_1, \ldots, d_n\right)$ be a solution to (2). Then we can define a solution s to constraint system C as $s\left(v_i\right) = d_i$. Indeed, let $\left(v_i, f\right) \in C$, then by definition of h_i, $s\left(v\right) = d_i \sqsupset f\left(s\right)$. This means that if F is monotonic and height of L is finite, then based on Kleene's fixed point iteration theorem, we can find the least solution to the constraint system. Furthermore, the theorem also offers us an algorithm for finding the solution.

It is easy to verify that if all the functions f in constraint system C are monotonic, then F is also monotonic. Let's note that the constraints defined in (1) are monotonic and so is the constant function that constraints the starting state, meaning that the constraint system we proposed to calculate $\mathcal{A}$ is solvable, presuming that the lattice it is defined on does have finite height.

As mentioned before, we could calculate the chain in Kleene's fixed point iteration to find the solution to the constraint system. However, there are also other approaches that are computationally cheaper. In the following, we will look at one of the simplest of them and show that computing the solution with it is feasible.

The solver we will analyse is the *round-robin* solver:

Algorithm 1: Round-robin solver for constraint systems on lattices.

```
begin
    foreach $v \in Var_C$ do
        $s[v] \leftarrow \bot$;
    end
    dirty $\leftarrow$ true;
    while dirty do
        dirty $\leftarrow$ false;
        foreach $(v, f) \in C$ do
            updated $\leftarrow s[v] \sqcup f(s)$;
            if updated $\neq s[v]$ then
                $s[v] \leftarrow$ updated ;
                dirty $\leftarrow$ true;
            end
        end
    end
    return $s$
end
```

The algorithm first initializes the potential solution with $\bot$ elements.

Then it checks for every constraint if it is satisfied – if it is, then $s[v] \sqsupseteq f(s)$, which directly implies that $s[v] = s[v] \sqcup f(s)$. f the constraint (v, f) is not satisfied, the value of v in the potential solution is updated with the upper bound of the current value and the value of f at the current potential solution.

Let $L = (D, \sqsubseteq)$ be a lattice that the constraint system C being solved with the round-robin solver is defined on. As the maximum number of times a value for a constraint system variable can be updated is the height of the lattice, then the outer loop of the algorithm cannot run more than $h \cdot |Var_C| + 1$ times, while the inner loop runs for $|C|$ times. Altogether, the maximum amount of times we have to evaluate the constraint function is upper-bounded by $h \cdot |Var_C| \cdot |C| + |C|$. Although this algorithm is efficient enough to be feasibly used in practice, there are faster algorithms that can be used to solve the constraint system and the round-robin algorithm serves here as a proof of the feasibility of a computation for solving the constraint system.

With this we have given a brief overview of abstract interpretation and how to compute analyses. In the latter, we did limit ourselves to single procedure programs. With some effort, this approach can be generalized to multithreaded programs involving procedure calls.

Next we will look into two key ideas the static detection of data races is based on.

4 Data Race Analysis

As discussed previously, it is possible that the same function of a driver is executed concurrently. Let *thread template* be the executable code reachable from one of the registered entry points of the driver. There can be any number of threads executing a single thread template at any point, corresponding to multiple calls to the same function of a device driver.

4.1 Lockset Analysis

Lockset analysis is a high-level idea about how to verify the absence of data races in a program. It relies on the assumption that the access to a shared variable should be governed by a lock.

Let us have two threads, T_1 and T_2 executing thread templates t_1 and t_2. Let's note that t_1 and t_2 do not have to be distinct.

Let S be a shared state between those two threads, containing variables that are accessible from both threads and L be a set of all locks that can be held by either thread.

Now, let O_T^v be a set of all read or write operations of variable $v \in S$ in thread T. Let's define operation *locks* : $O_T^v \to 2^L$, that for a read or write operation of variable v in T returns a set of all the locks that thread T holds when that operation is performed.

If set

$$\left(\bigcap_{o \in O_{T_1}^v} \mathit{locks}\,(o) \right) \cap \left(\bigcap_{o \in O_{T_2}^v} \mathit{locks}\,(o) \right)$$

is not empty, then it means that there exists $l \in L$ such that every read or write operation of variable v in threads T_1 and T_2 is protected by l and as such, there is no chance for a data race. If such a lock exists for every variable $v \in S$, we can soundly say that there is no possibility of a data race between threads T_1 and T_2 executing thread templates t_1 and t_2. Note that this approach can be easily generalized for n threads and m thread templates.

Let's note that although this analysis is sound (we will not verify a program that has a possibility of a data race), it is not precise, as there might be other ways than locking to make sure that no data race can take place. This idea was first applied in [7].

4.2 The Happens-Before Relation

Happens-before relation R is a partial-ordering of the statements, such that if a and b are statements then $a, b \in R$ if and only if a is executed before b. The concept was first introduced by Leslie Lamport [11], under the name happened-before.

For a to be executed before b, either a has to be a statement before b in the same thread template, there must be a statement c such that a is executed before c and c is executed before b or there must be a synchronization event that is sufficient to say that a was executed before b.

As an example of a synchronization event, let's consider spawing a thread. Let t_1 and t_2 be two thread templates. Let t_1 be a thread template where a variable x is read and then a thread from template t_2 is spawned. Let t_2 be such a thread template where a value is assigned to x. The thread spawning functions here as a synchronization event – if the thread T_1 runs the template t_1 and spawns the thread T_2, then the read of x in t_1 happens before the assignment in t_2.

The key insight concerning data races is that if we can establish happens-before relationship between statements a and b, then they cannot race against each other.

4.3 Field Survey

We will now give an overview of notable tools that detect data races in device drivers.

4.3.1 KernelStrider

KernelStrider[23] is a dynamic analyser front-end for kernel space programs. It collects information about the execution of a device driver and forwards it to user-space. This makes it possible to use programs developed to detect data races in any C program to be used with device drivers. KernelStrider aims to make use of ThreadSanitizer[22], a dynamic data race analyser by Google for user-space, that uses both lockset analysis and happens-before. Another user-space dynamic analyser that KernelStrider output could be used with is Helgrind, [1, 14] which utilizes happens-before.

4.3.2 RaceHound

RaceHound[10] is a dynamic analyser for the Linux device drivers. It borrows heavily from DataCollider [8], an analyser for Microsoft drivers. RaceHound places random software breakpoints to the driver under test, then finds out which memory address was accessed and attaches a hardware breakpoint to that memory address and stops the thread for a while. When the hardware breakpoint is triggered by another thread, a data race has taken place.

The main benefit of this approach is the relatively low overhead of around 5% to the running time of the program under test. This approach is precise, but not sound.

4.3.3 Locksmith

Locksmith[17] is a static analyser of data races in C programs, including device drivers. Locksmith was one of the earliest static analysers available for the Linux device drivers and it validated the lockset based approach for static analyses.

For practically inclined, the authors of Locksmith share their experience with static analysis of data races, describing the importance of identifying thread-local variables as a preprocessing step and of modelling fields of the structs field-sensitively and lazily[18].

4.3.4 Checkmate

Checkmate[9] is a generic static analyser of Java bytecode. It offers support for analysing different runtime properties of a multi-threaded program running on JVM, among them data race detection.

Checkmate stands out by making extensive use of happens-before relation and not relying on the lockset analysis for detecting data races. The happens-before used by Checkmate over-approximates Java Memory Model.

4.3.5 Whoop

Whoop[6] is a static analysis tool for finding data races in the Linux device drivers.

Whoop is a sound tool, making use of the lockset analysis. Whoop is based on the observation that data races happen between two specific threads and as such, it is enough to verify that there is no possibility of data race between any pair of threads, instead of considering all the threads at the same time.

Whoop is meant to be used as part of a bigger tool-chain, where it functions as a preprocessing step to Corral, a bug-finder by Microsoft. During the preprocessing step, Whoop produces a sound model of the driver that can then be processed further by Corral.

The fitting together of a quite complex pipeline of tools to provide the analysis is a remarkable result of the project behind Whoop.

As of now, Whoop supports classical mutexes and spin-locks, but does not cover rest of the primitive locking options that are usable in the Linux device drivers.

4.3.6 SDV and SLAM

SDV is a static analyser by Microsoft for Windows device drivers.

As mentioned before, majority of the failures in Windows XP were caused by issues connected to device drivers. To solve this issue, Microsoft Research team started working on a way to verify the correctness of device drivers in early 2000s. The result of their work is Static Driver Verifier (SDV) that tries to verify a device driver during compile time.

The key part of SDV is SLAM [3], a tool that allows one to specify rules that must hold for device drivers and then statically check if they do for a driver during compile time.

The main use case for SLAM in SDV is to confirm that drivers use the Windows driver API correctly.

SDV has been widely successful, with under 10% of reported issues being false reports [3, 74] and 97% of the runs resulting in either verification of the driver or finding a bug.

Although SDV does not focus on detecting data races, its success has shown that static analysis can be a viable approach for driver verification and it has had a big influence on the field of static analysis of device drivers.

5 Region Analysis of Space and Time

One of the key challenges for static analysers based on the abstract interpretation is to choose abstract domains in a way that minimizes the loss of precision stemming from over-approximation. The loss of precision causes static analysers to emit false-positives and it is clear that even if one can guarantee the soundness of the underlying analysis, there is a threshold of incorrect warnings above which the results of the analysis contain too much noise to be helpful to the end user.

We will now introduce two analyses that, when combined with other analyses, help to increase the precision of a static analyser that detects data races. Both of these analyses are implemented in the static analyser of *Goblint* that we will discuss further in the next section.

5.1 The Original Region Analysis

First of the analyses is the *region analysis*. The key idea behind this analysis is to try to partition the memory used by the program under analysis into disjoint regions. Based on this information we can deduce that memory accesses to different regions cannot data race. This does not exclude the possibility of a data race inside a region, but it means that for every region there can be a separate locking mechanism.

As an example, consider a hash table using chaining to handle hash collisions that has a lock for each of the buckets.

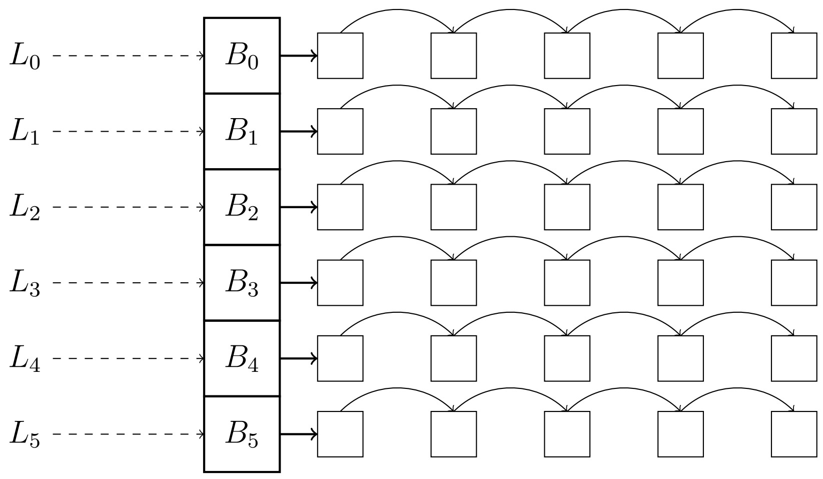


Figure 9: Separate chaining hash table with each bucket being protected by a lock

If the hashes of the keys k_1 and k_2 are different, then accessing them in the hash table cannot cause a data race. On the other hand, if k_1 and k_3 produce the same hash, then there is a real risk of a data race and Goblint cannot exclude this possiblity without further information. It would be enough to know that in both cases, the lock protecting bucket corresponding to this hash was held during the access.

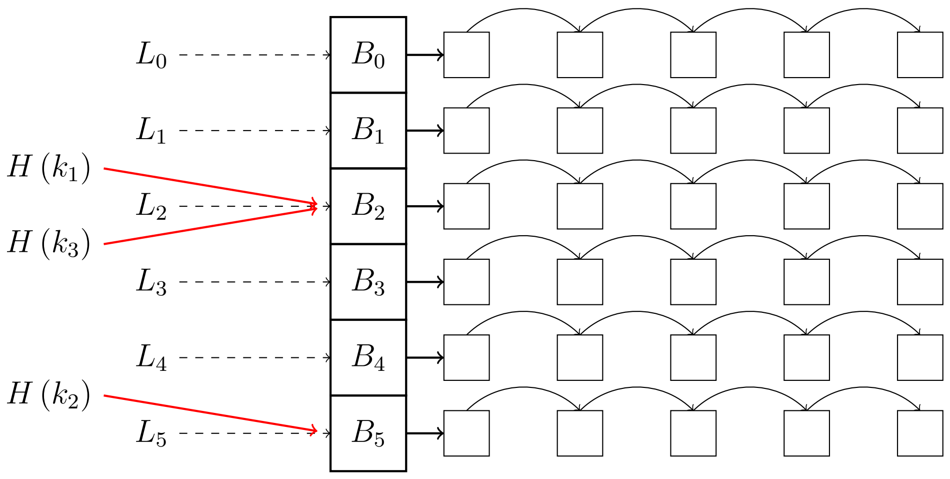


Figure 10: Separate chaining hash table

If we can guarantee that when adding an element for key k_1 and for key k_3, we are accessing different regions of the memory, then we can deduce that these additions cannot cause a data race.

For a hash table, the most obvious partition for the regions would be to separate the buckets of the hash table. This can be done if we can guarantee that the same memory location is not accesible from different buckets.

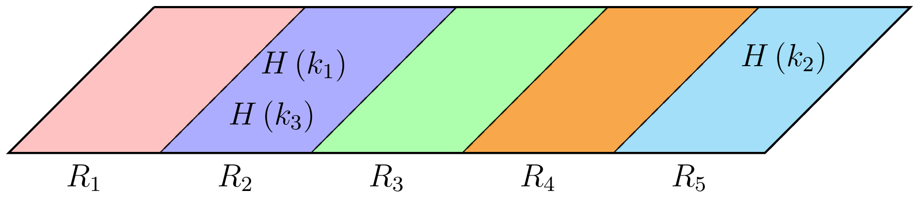


Figure 11: Memory partition of a hash table

More information regarding region analysis can be found in [20], while the analysis tracking the equalities between hash table keys is described in [21].

5.2 Regions in Time

Device drivers contain not only disjoint regions of memory, but also disjoint time intervals.

Consider the following simplified device driver:

```
static int i = 0;
static int j = 0;

static int file_open(file* f, ...)
{
    printk( "Opening device \n");
    file -> private_data += 1;
    j += 1;
    return 0;
}

static int file_close(file* f, ...)
{
    printk( "Closing device \n");
    file -> private_data -= 1
    j -= 1;
    return 0;
}

int a(file f, ...)
{
  j += 1;
  return 0;
}

int b(file f, ...)
{
  j += 1
  return 0;
}

struct file_operations f_ops = {
  .open = file_open,
  .release = file_close,
};

int init(...)
{
  publish_file_operations(f_ops);
  i += 1;
  return 0;
}
```

```
int exit(...)
{
  i -= 1;
  return 0;
}
```

As mentioned previously, `init` and `exit` functions are used by the kernel to register and deregister a device. After publishing the file operations we can no longer assume that the functions of the driver are called by only one thread and hence the static analyser should detect a possible data-race (given there is no additional information) between the operations on the variable i in functions `init` and `exit`. At same time, the Linux kernel does not allow a call to the `exit` function until the call to the `init` function has completed and all the opened files are released.

Our aim is to have an analysis that knows about such guarantees and can make use of the information the guarantees provide. To do so, inspired by the happens-before relation, we extend the region analysis to cover time in addition to space. Let the example driver have two memory regions, A and B, and let `i` belong to the region A and `j` to the region B.

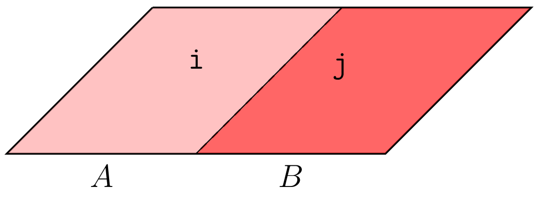


Figure 12: Memory partition for the example driver as done by region analysis

Unfortunately, this does not allow us to eliminate the data race between the assignments to `i` in `init` and `exit` functions. However, if we would enhance the region analysis with an additional dimension that would roughly correspond to time, we could divide the memory as seen on the following illustration.

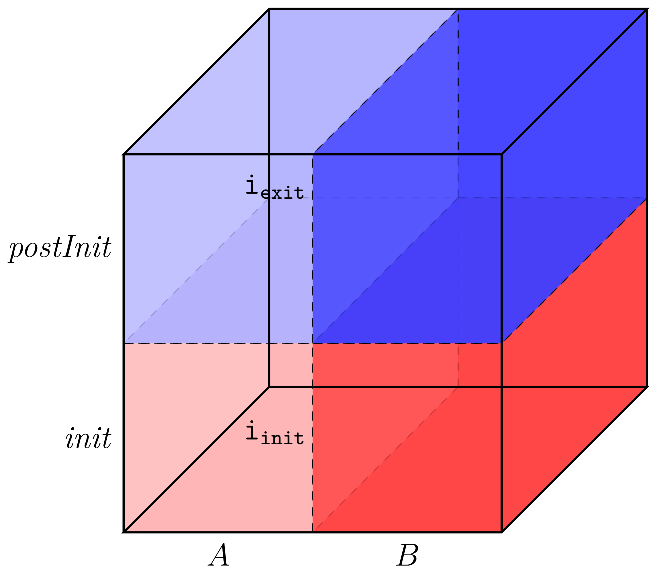


Figure 13: Memory partition with addition of time dimension

Here, *init* and *postInit* are both *time regions*. This lets us exclude the possibility of race between these two stamements.

To do this more generally, we need to consider the information that we track for each read and write access. We will divide the tracked information into the *left* and *right* side of an implication: if two accesses may collide, then certain safety conditions must hold to avoid a race condition. On the left side, information regarding the region of the statement is stored as a set of sets, C, such that every element $I \in C$ describes an intersection of regions. This allows an extensible and modular framework, such that we can add an analysis to provide a different means of partitioning accesses without being concerned with the already existing ones.

Conceptually, every element of C describes a possible region for the access. So if we know that the set C for x_a is in

$$\{\{A, \mathit{init}\}, \{B, \mathit{postInit}\}\}$$

then we can say that the statement is either located in region A and region *init* or in region B and region *postInit*.

In our example driver, the C for the increment statement of i in the *init* function would be

$$\{\{A, init\}\}\,.$$

On the right side, information regarding possible reasons about why a data race cannot take place with another read or write statement is stored in a set M. For example, M could contain the set of locks held at the time the statement is executed or information about the guaranteed uniqueness of the thread running the procedure that contains the statement.

When deciding on the possibility of a data race between statements a and b, we use two predicates, L and R to evaluate left and right sides of these statements.

The first of these predicates, L, is true if it is possible that the statements share the same region. It is defined as follows:

$$L\left(C_a, C_b\right) = \exists I \in C_a,\, \exists J \in C_B\left(I \subseteq J \vee J \subseteq I\right).$$

If $L\left(C_a, C_b\right)$ evaluates to false, it means that the two statements cannot share a common region and hence there cannot be a data race.

The second predicate, R evaluates the sets M_a and M_b and returns true if there is something that guarantees that the statements a and b cannot race. For example, it could be when performing both of these statements, lock l must be held or in case of $a = b$, there is a guarantee that there can be only one thread running that executes the statement.

Equipped with those definitions, we can guarantee that statements a and b cannot race if the following implication holds.

$$L\left(C_a, C_b\right) \implies R\left(M_a, M_b\right) \tag{3}$$

Returning to our example driver, one can see that using the enhanced region analysis, there cannot be a race between i_{init} and i_{exit}. Indeed, as

$$\{\{A, init\}\} \cap \{\{A, postInit\}\} = \emptyset$$

then $L\left(C_{i_{init}}, C_{i_{postInit}}\right)$ is false and the implication (3) holds for i_{init} and i_{exit}.

As an example of right-hand sides excluding a data race, if we take both a and b to be i_{init}, then when evaluating

$$R\left(M_{i_{init}}, M_{i_{init}}\right)$$

we can take into account that the Linux kernel does not allow one to register the same driver more than once and as such, we can guarantee that the thread running the *init* function is unique.

Based on that information $R(M_{i_{init}}, M_{i_{init}})$ holds and so does the implication (3).

One might have noticed that our example drivers have two extra functions, a and b. Let's now suppose that a similar property holds for a and b. During the lifetime of the driver, we can only call a once and b once and the call to a must happen before the call to b.

Let the following be the regions of the program, enhanced with the extra dimension, taking into account only the division by a and b.

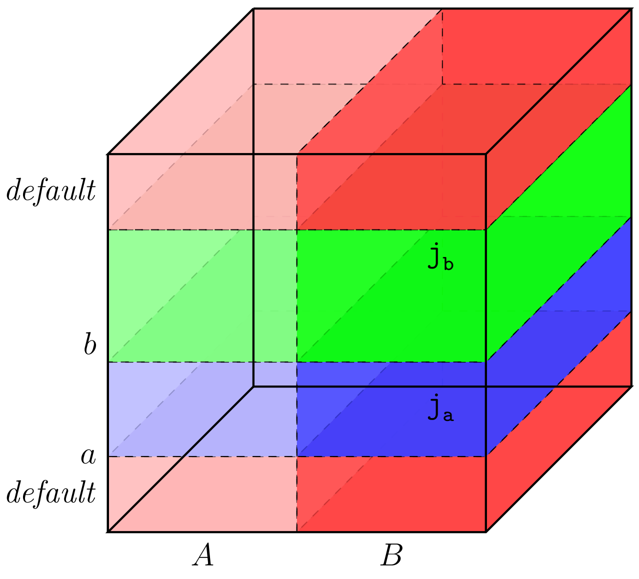


Figure 14: Memory partition with addition of time dimension by a and b.

If we want to combine this time division with the one we had for functions `init` and `exit`, we could find the Cartesian product of the two and use elements of that set as time regions. The resulting partition is shown in the next figure.

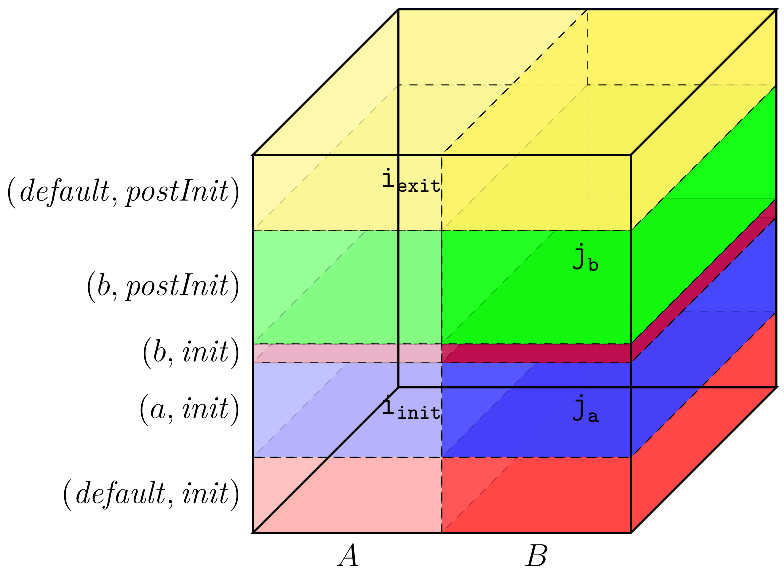


Figure 15: Memory partition with addition of combined time dimension

Here, the set C for the increment statement of i in the *init* function would be

$$\{\{A, (init, default)\}, \{A, (init, a)\}, \{A, (init, b)\}\}$$

and for the decrement statement of j in b it would be

$$\{\{B, (init, b)\}, \{B, (postInit, b)\}\}.$$

It is easy to see that this approach does not scale too well – in the worst case the computational difficulty will grow exponentially with each component of the Cartesian product that form the time partition.

However, notice that when we evaluate the left sides of the two statements, the time dimension only has an effect if memory regions of the statements overlap; if the memory regions are disjoint, the predicate L is already false. With that in mind, we can construct the time regions for the memory region A by combining only the relevant time partitions for the variables in the region A. For our running example, the partition would look as follows.

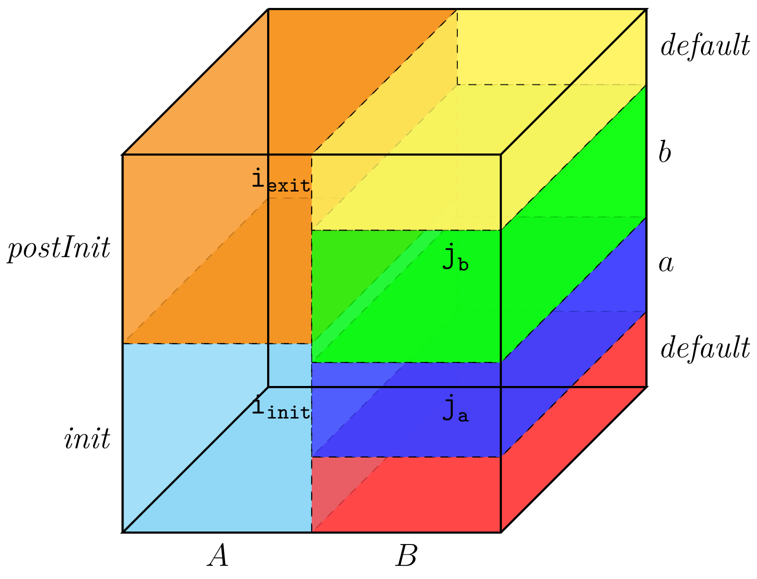


Figure 16: Memory partition with local time partition.

With this partition, the set C for the increment statement of i in the *init* function would be

$$\{\{A, \mathit{init}\}\}$$

and for the decrement statement of j in b it would be

$$\{\{B, b\}\}\,.$$

This approach scales well if we are able to divide memory into small areas. More importantly, the finer granularity of this approach lets us support more specific guarantees. Once again, let's consider the functions `file_open` and `file_release` of the example driver we have used in this section.

```
static int file_open(file∗ f, ...)
{
    printk( "Opening device \n");
    file -> private_data += 1;
    j += 1;
    return 0;
}

static int file_release(file∗ f, ...)
{
    printk( "Closing device \n");
    file -> private_data -= 1
    j -= 1;
    return 0;
}
```

It is guaranteed that the file f can only be closed once for each time it is opened and the operations have to alternate. This means that the operations on `file -> private_data` cannot race in functions `file_open` and `file_release`. At the same time, it is very possible that there is a race between the operations on `j` in `file_open` and `file_release` – it is possible that the file f_1 is being opened at same time as f_2 is being closed.

If we are able to separate the memory region that only contains `file`, then we can partition this further by dividing the time dimension to *default*, *file_open* and *file_release*, while not applying the same division of time dimension to the memory region that contains `j`.

6 Implementation in Goblint

In this section, we give describe the implementation time partitions in the *Goblint* analysis framework. Goblint is a static analyser for detecting data races in C programs with the focus on Linux device drivers that has been developed in the University of Tartu and in the Technical University of Munich for over 10 years [28, 26, 2, 27]. As the original work presented in this thesis builds on top of what is done in Goblint, we will first give the necessary background information that will hopefully enable the reader to put that in context.

6.1 The Anatomy of Goblint

Goblint is a constraint system based analysis tool that performs abstract interpretation, building on top of the theoretical ideas described in the previous sections. It is written in OCaml and is publicly available on Github.[4]

Goblint takes a C program as an input and uses the CIL framework [13] to process it into an intermediate form that can then be converted into a control flow graph (*CFG*). As the next step, the enabled *analyses specifications* are combined and a constraint system that corresponds to the combined analyses and CFG of the program is produced. The generated constraint system is then solved with a generic constraint system solver. Finally, the output is mapped to the original C program and displayed to the user in a suitable format.

[4]`https://github.com/goblint/analyzer`

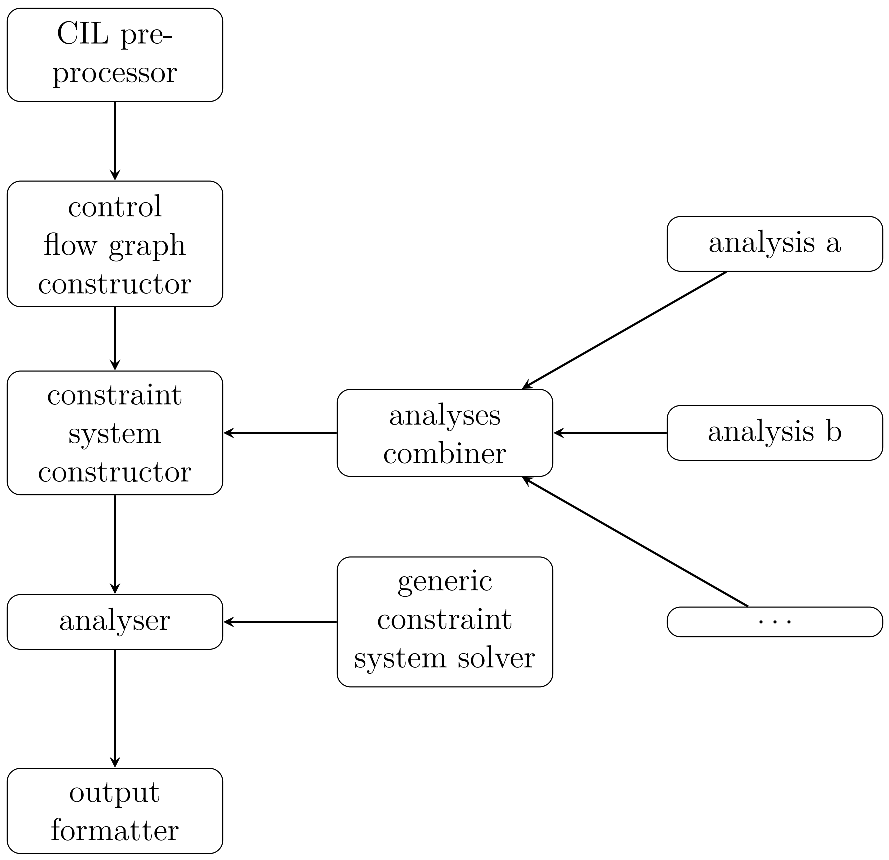

CIL pre-processor
control flow graph constructor
constraint system constructor
analyser
output formatter
analyses combiner
generic constraint system solver
analysis a
analysis b
...

Figure 17: The structure of Goblint.

The most interesting components of Goblint are the provided solvers for the constraint systems and the wide array of analyses. We will be focusing on the structure of an analysis, using the example of the mutex analysis, which makes use of the previously discussed ideas about lockset analysis. This will hopefully both introduce one of the key analysis for Goblint and give a high-level idea of how an implementation of an analysis looks like. The signatures we introduce are simplified versions of the ones present in the Goblint implementation.

The solvers implemented in Goblint do not fall with the scope of this thesis. A thorough overview can be found from [2, 26].

As previously described, for each analysis, we need an abstract domain to perform the abstract interpretation on. In Goblint, all the domains implement interface `Lattice`. The functions *join* and *meet* correspond to the binary least upper bound and greatest lower bound operations.

```
module type Lattice =
sig
  type t
  val leq: t →t →bool
  val join: t →t →t
  val meet: t →t →t
  val bot: unit →t
  val is_bot: t →bool
  val top: unit →t
  val is_top: t →bool
  ...
end
```

The type `t` in OCaml module type leaves the type abstract. From here on, the types written in italics are built-in types in OCaml.

In case of the lockset analysis, the domain is a reversed set domain of all the possible locks. That is, the elements in the domain are sets of locks in the program under analysis and join operation is defined as the intersection of two sets and meet as union of two sets. The reasoning behind the set domain being reversed is that we are interested in the set of all the locks that must be held at a specific statement. In the situation where we have to take an upper bound of the two locksets (say, after two branches of an if-else statement merge), we hence want it to only consist of locks that were in both of the locksets.

Analysis in Goblint must implement the module type `Spec`. Following is the simplified version of it.

```
module type Spec =
sig
  module D : Lattice
  module G : Lattice

  val name : string

  val startstate : varinfo →D.t
  val exitstate : varinfo →D.t
  val otherstate : varinfo →D.t

  val part_access: (D.t, G.t) ctx →exp →varinfo option →bool →(Access.LSSSet.t ∗ Access.
      LSSet.t)

  val query : (D.t, G.t) ctx →Queries.t →Queries.Result.t

  val assign: (D.t, G.t) ctx →lval →exp →D.t
  val branch: (D.t, G.t) ctx →exp →bool →D.t
  val body : (D.t, G.t) ctx →fundec →D.t
  val return: (D.t, G.t) ctx →exp option →fundec →D.t
  val special : (D.t, G.t) ctx →lval option →varinfo →exp list →D.t

  ...

end
```

Here the underscored types are the ones defined in the CIL library.

Each analysis includes two lattices, lattice D will contain the abstract states and lattice G the abstract values for global variables.

The type `ctx` with type parameters $(D.t, G.t)$ encapsulates both the local and the global state, offering access to helper functions on them.

The functions `startstate`, `exitstate` and `otherstate` provide an initial abstract state for a thread depending on whether the thread starts an initialization function (in the context of device drivers, the function that registers the driver and exposes the callback functions), a cleanup function (where deregistration happens in device drivers) or is any other function that can be used to spawn a new thread (all the file operations).

The functions take as argument information about the function declaration, enabling Goblint to differentiate between different file operations. In case of mutex analysis, all the functions always return empty locksets.

The `part_access` function enables one to partition accesses into disjoint groups – if two accesses to the variable x are in different partitions, then they cannot race. In addition, it also associates a set of locks held with the partition. This function makes it possible to use the idea of dividing access information to the left and right side.

In case of the mutex analysis, this function returns empty sets, as it does not make use of partitioning, but plays an important role in region analysis.

The `query` function enables the communication between different analyses that have been combined into one and are being performed at the same time – an analysis must be able to answer the query based on the current context. This allows an analysis to avoid duplication and allows one to design more modular analyses consisting of sub-analyses. More information regarding the query system of Goblint can be found in [2, 113].

The functions `assign`, `branch`, `body` and `return` are functions from abstract state to abstract state, for assignment statements, branching statements and for entering and exiting from function calls. They correspond to the $[\![s]\!]^*$ family of functions previously introduced.

In the implementation of mutex analysis, those transfer functions do not actually change the abstract state, as none of the statements influence the set of held locks. However, they do register accesses to variables, for example, an access is registered for both the left and right hand side of an assignment with further distinction that the access to the left hand side is a write access.

The `special` function handles functions for which the source code is not available for Goblint to analyse. In the mutex analysis, this function modifies the abstract state – function calls to functions such as `__raw_lock_unlock` and `_lock_kernel` remove and add mutexes to the set of held locks.

6.2 Implementation of Time Partitions

As part of this thesis, the region analysis was extended to include the time dimension in addition.

A new concurrency domain was added that improves the way Goblint handles detection of processes that are guaranteed to run only in one thread at a time. A way to append the time partition information to the left side of the access was added with a method to transform the time partition that can be used inside transfer functions.

The implementation makes vast use of the `part_access` method. A remaining technical difficulty is that the concurrency domain is ingrained in the base analysis, one of the oldest analysis in Goblint, and Goblint does not have a preproccessing phase. Therefore, the complexity added to the already complex base analysis is quite extensive.

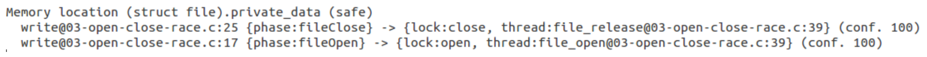
I hope to solve this issue before requesting the changes to be merged into the mainline. Meanwhile, a fork of Goblint containing these changes is available here: `https://github.com/vootelerotov/goblint-fork`

For a concrete example, let's consider the following device driver:

```
pthread_mutex_t open = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t close = PTHREAD_MUTEX_INITIALIZER;

static int file_open(struct inode *inode, struct file *file)
{
  pthread_mutex_lock(&open);
  file->private_data=1;
  pthread_mutex_lock(&open);
  return 0;
}

static int file_release(struct inode *inode, struct file *file)
{
  pthread_mutex_lock(&close);
  file->private_data=NULL;
  pthread_mutex_lock(&close);
  return 0;
}

...
```

Previously, Goblint would have seen `private_data` of `file` as unsafe. At the same time, Linux kernel guarantees that a file cannot be released until it has been opened and it can be opened again only after it has been released. This guarantee has been incorporated into Goblint by partitioning the time for `file` argument into partitions *fileOpen*,*fileClose* and *default*.

Since accesses to the `private_data` are in different regions, Goblint is able to rule out a possible data race and deem the location safe, as can been seen from the Goblint's output in Figure 18



```
Memory location (struct file).private_data (safe)
  write@03-open-close-race.c:25 {phase:fileClose} -> {lock:close, thread:file_release@03-open-close-race.c:39} (conf. 100)
  write@03-open-close-race.c:17 {phase:fileOpen} -> {lock:open, thread:file_open@03-open-close-race.c:39} (conf. 100)
```

Figure 18: Snippet from Goblint output.

6.3 Evaluation

Evaluating the effect of the enhanced region analysis on the set of character devices used as benchmark suit for Goblint gave the following results.

Benchmark suite				
	Indirect races		Direct races	
Driver	Region	Enhanched Region	Region	Enhanched Region
apm-emulation.c	27	27	24	19
applicom.c	16	15	8	7
bsr.c	19	0	10	2
dtlk.c	30	23	19	19
efirtc.c	0	0	0	0
genrtc.c	0	0	2	2
hangcheck-timer.c	12	12	0	0
hpet.c	292	133	27	25
ipmi_devintf.c	197	117	8	8
ipmi_msghandler.c	481	481	273	273
ipmi_poweroff.c	142	50	11	11
ipmi_watchdog.c	245	224	13	13
lp.c	85	85	8	8
mem.c	0	0	5	5
misc.c	6	6	10	9
nvram.c	0	0	1	1
pc8736x_gpio.c	195	0	2	2
ppdev.c	129	122	14	14
random.c	65	65	40	40
raw.c	264	0	3	2
rtc.c	20	1	4	3
scx200_gpio.c	188	0	1	1
tlclk.c	86	18	8	7
toshiba.c	0	0	0	0
ttyprintk.c	193	193	3	3

Figure 19: Results of benchmarking region analysis with and without time partitioning

We divide the potential data races into two categories. We say that the potential race is *direct* if Goblint deems it unsafe based on the direct access present in the source code Goblint is analysing. For example, the direct access could be a variable assignment. An *indirect race* is reported when there is a possibility of a race at a location, when one considers functions for what Goblint does not have the source code for. This can happen if driver exposes some of its states using a function defined in the Linux kernel.

In the Figure 6.3 the columns denoted as *Region* contain results of the analysis of the benchmark suite performed by Goblint with the space based region analysis active, while the column denoted as *Enhanched Region* are results that were achieved by additionally adding the time partitioning. The numbers indicate real data races or false-positives found by Goblint.

As seen by the results of the benchmark, the effect the time partition added to the region analysis depends heavily on the driver. In some drivers, it enables Goblint to deem quite a lot of previously unsafe locations to be safe, reducing the amount of false-positives. Notable examples are `impi_poweroff`, `raw.c`, `scx200_gpio.c`, `tlclk.c`, where over half of the potential indirect races were eliminated. The results for the driver `bsr.c` stood out for noticeable improvement in precision for both indirect and direct races.

In most of the drivers, the positive effect can be attributed to the exclusion of races between `init` and `exit` functions.

It is worth noting that the motivation for introducing the partitioning by time to Goblint came from analysing the common issues on a subset of those very same drivers. However, the fact that there were other drivers, outside the subset of the benchmark drivers that were manually analysed that benefited from the addition of the extra dimension, leads me to believe that there is a reasonable likelihood that the benefits are not limited to this specific set of drivers. Out of the 6 drivers that were mentioned as notable examples, only one, `bsr.c` was part of the manually reviewed drivers.

7 Conclusion

The goal of this thesis was to improve the precision of Goblint, a static analyser for detecting data races in Linux device drivers. I focused on the false-positives found during the review of Goblint's performance on a benchmark suite.

The region analysis of Goblint, which divides memory to disjoint regions to rule out data races in certain situations, was enhanched with an additional dimension corresponding to time. This allows Goblint to eliminate a possibilty of data race on the basis of happens-before relationship.

In addition to implementing the enhanched analysis in Goblint, I also provided a theorectical overview of the said enhanchment. Background knowledege regarding Linux device drivers, abstract interpretation, static analysis of data races and Goblint is provided, in the hope of making the thesis self-contained. Lastly, we introduce other notable tools that analyse data races in device drivers.

As a result of the enhancement, the percision of Goblint on the mentioned benchmark suit was improved. Notably, in 6 out of 25 drivers, over half of the potential races where deemed to be false-positives by the enhanched region analysis.

For futher improvements in precision I hope to introduce further partitions of time. Before that, however, further work on benchmarking is required, to both see if the improvements seen on the benchmark set of device drivers can be generalized to all device drivers and to find false-positives that can be eliminated by further paritioning of time. The technical challenges are to provide a way to introduce new time partitions conveniently, for example through a configuration file, and to increase the modularity of the implementation.

In addition to that, it would be very interesting to see how such an analysis fares in conditions where the happens-before guarantees are formally specified, for example by Java Memory Model.

References

[1] Helgrind. `http://valgrind.org/docs/manual/hg-manual.html`.

[2] Kalmer Apinis. *Frameworks for analyzing multi-threaded C.* PhD thesis, Technische Universität München, Munich, 2014.

[3] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A Decade of Software Model Checking with SLAM. *Commun. ACM*, 54(7):68–76, July 2011.

[4] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, 1977.

[6] Pantazis Deligiannis, Alastair F. Donaldson, and Zvonimir Rakamarić. Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2015, pages 166–177, Washington, DC, USA, 2015. IEEE Computer Society.

[7] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[8] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-race Detection for the Kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[9] P. Ferrara. Checkmate: A Generic Static Analyzer of Java Multithreaded Programs. In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 169–178, November 2009.

[10] Nikita Komarov. On the Implementation of Data-Breakpoints Based Race Detection for Linux Kernel Modules. 2013.

[11] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[12] V.S. Mutilin, E.M. Novikov, and A.V. Khoroshilov. Analysis of typical faults in Linux operating system drivers. *Proceedings of the Institute for System Programming of RAS*, 22:349–374, 2012.

[13] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.

[14] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[15] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[16] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 305–318, New York, NY, USA, 2011. ACM.

[17] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: Context-sensitive Correlation Analysis for Race Detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 320–331, New York, NY, USA, 2006. ACM.

[18] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: Practical Static Race Detection for C. *ACM Trans. Program. Lang. Syst.*, 33(1):3:1–3:55, January 2011.

[19] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming Device Drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 275–288, New York, NY, USA, 2009. ACM.

[20] Helmut Seidl and Vesal Vojdani. Region analysis for race detection. In *Proceedings of the 16th International Static Analysis Symposium*, SAS'09, pages 171–187, Berlin, Heidelberg, 2009. Springer-Verlag.

[21] Helmut Seidl, Vesal Vojdani, and Varmo Vene. A smooth combination of linear and Herbrand equalities for polynomial time must-alias analysis. In *Proceedings of the 16th International Symposium on Formal Methods*, FM'09, pages 644–659. Springer-Verlag, 2009.

[22] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[23] Eugene Shatokhin. Kernel Strider. `https://github.com/euspectre/kernel-strider`.

[24] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 207–222, New York, NY, USA, 2003. ACM.

[25] Linus Torvalds. Linux 4.0. `https://github.com/torvalds/linux`.

[26] Vesal Vojdani. *Static Data Race Analysis of Heap-Manipulating C Programs.* PhD thesis, University of Tartu, Tartu, 2010.

[27] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static Race Detection for Device Drivers: The Goblint Approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, New York, NY, USA, 2016. ACM. To appear.

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

Appendecies

A Example driver

```
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */
#include <linux/init.h> /* Needed for the macros */
#include <linux/cdev.h> /* cdev struct */
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/slab.h>
#include <linux/gfp.h>
#include <linux/kdev_t.h>
#include <linux/mutex.h>
#include <linux/delay.h>

static struct cdev my_cdev;

static int i;

static dev_t devid;

static int file_open(struct inode *inode, struct file *file)
{
    //printk(KERN_INFO "Opening file \n");
    return 0;
}

static int file_release(struct inode *inode, struct file *file){
    //printk(KERN_INFO "Closing file \n");
    return 0;
}

static ssize_t file_read(struct file *fp, char __user *buf, size_t nbytes, loff_t *pointer){
    size_t buf_size;
    size_t written;
    size_t bytes_returned;
    size_t maximum_number_of_bytes_written;
    char *buffer;
    i++;
    printk("Increasing i, new value: %d \n",i);
    buffer = (char *)kmalloc(nbytes,GFP_KERNEL);
    buf_size = sizeof buffer;
    maximum_number_of_bytes_written = buf_size < nbytes ? buf_size : nbytes;
    written = snprintf(buffer,nbytes > buf_size ? buf_size : nbytes,"%d%c",i,'\n');
    //printk("Asked for %zu, got: %zu, write: %zu \n",nbytes,buf_size,written);
    bytes_returned = written < maximum_number_of_bytes_written ? written :
        maximum_number_of_bytes_written;
    copy_to_user(buf, buffer, bytes_returned );
    kfree(buffer);
    return bytes_returned;
}

static ssize_t file_write(struct file * fp, const char __user * ub, size_t s, loff_t *pointer){
    i--;
    printk("Decreasing i, new value: %d \n",i);
    return s;
}

static struct file_operations my_file_ops = {
        .owner = THIS_MODULE,
        .write = file_write,
        .read = file_read,
        .open = file_open,
        .release = file_release,
        .llseek = no_llseek,
};

static int __init hello_start(void){
    int rc;
    printk(KERN_INFO "Loading driver test\n");
    i = 0;
    printk(KERN_INFO "Setting i to 0\n");
    rc = alloc_chrdev_region(&devid, 0, 1, "mydriver");
    if (rc < 0){
        printk(KERN_INFO "Failed to alloc region");
    }
    printk(KERN_INFO "Allocated number \n");
    printk("Major: %d, minor: %d \n", MAJOR(devid),MINOR(devid));
    cdev_init(&my_cdev, &my_file_ops);
    cdev_add(&my_cdev, devid, 1);
    return 0;
}

static void __exit hello_end(void){
    printk(KERN_INFO "Goodbye Mr.\n");
    cdev_del(&my_cdev);
    unregister_chrdev_region(devid, 1);
}

module_init(hello_start);
module_exit(hello_end);
```

B Test script

```
#!/bin/bash

 for i in ‘seq 1 10000‘;
        do
                echo h > /dev/my-driver
    head -c 1 /dev/my-driver >> /tmp/log1.txt
                echo "\n" >> /tmp/log1.txt
        done
```