## Introduction

Threads are used by programmers in order for their programs to do multiple tasks concurrently. They greatly decrease the execution time of a program, but their use creates a different problem. A thread may write to a memory address while at the same time one or more other threads are writing or reading that same memory address and may lead to unexpected results for the program. This is what we call a “data race”. Therefore, programmers have to ensure that their programs have no data races in them.

How exactly do they deal with data races? Usually, this is possible with the use of locks. A lock is a mechanism that, once acquired by a thread, it permits that one thread to access a shared location, while also preventing the other threads from accessing it at the same time. Once he thread finishes reading/writing in that location, it releases the lock and another random thread acquires the lock.

Now that we established how are these data races going to be removed, how are they going to find them when they occur? Running the program and detecting the races could work if it is small enough. However, in larger programs this would be nigh-impossible to do and too time-consuming. We can analyze and check the source code directly without actually running the program, thus saving time. This is called “static analysis”. So, programmers need a program that does exactly that and that’s what Locksmith is for.

## Locksmith

Locksmith is a tool that performs static analysis in order to detect data races in C programs. It tries to predict which memory addresses are prone to have data races by reading the program in compile time. Locksmith analyzes multithreaded programs that uses locks to protect shared memory locations and prevent concurrent reading and writing in them. It finds each shared memory address and checks which locks protect that location, if any. If not, then we have a data race in that location.

So, how exactly does Locksmith work? Well, Locksmith’s algorithm is divided in a number of phases. These phases are as follows:

**Typing phase**
Locksmith reads the entire program and stores its control flow. All information about the control flow is represented in the code by phi (φ) structs. Information about the global variables is represented by rho (ρ) structs.

**Shared phase**
In this phase, Locksmith finds all variables that could be used simultaneously among the threads. These variables/memory addresses are the ones who need to be checked

for data races. Other variables, such as thread local variables, are not prone to data races and thus are excluded from this analysis.

**Linearity phase**
Checks if the locks used are linear or non-linear. If a lock is non-linear, it means that it could represent multiple locks and Locksmith cannot know which lock is acquired in this case.

**Lock State phase**
Locksmith finds the state of each lock after each statement of the program i.e. if a lock is acquired at any point or not.

**Guarded-By phase**
In this phase, Locksmith checks each shared variable/memory address which locks are held each time they are accessed. More specifically, it initializes guards in each dereference of shared variables which stores a correlation of a variable with a corresponding lock, if any. Then, for each guard, it propagates through the program flow backwards until it reaches the main() function, possibly creating more correlations and guards on its way. It ends once every correlation has propagated to the main() function. After this phase, Locksmith knows where each variable acquires or releases a lock at any point, so it knows all the potential data races.

**Escapes phase**
Here, Locksmith completes the work done in Linearity Phase.

**Races phase**
Prints whether a variable could have a potential data race as a warning, along with that variable's references.

Now that we know how Locksmith works, how exactly does Locksmith shows the data races? For that, we will need an example like the one below.

Locksmith output

Now, consider the following program:

test.c

```
#include <pthread.h>

pthread_mutex_t lock1, lock2, lock[2];
float shared_var;
int partly_unprotected;
float non_linear;
```

```
int many_refs;
int completely_unprotected;

void *my_func(void *arg) {
  int local_var = 4, local_var2;
  pthread_mutex_lock(&lock1);
  shared_var++;
  many_refs += 5;
  pthread_mutex_unlock(&lock1);
  pthread_mutex_lock(&lock2);
  partly_unprotected = local_var;
  local_var += many_refs;
  pthread_mutex_unlock(&lock2);
  many_refs = 1;
  local_var = partly_unprotected;
  completely_unprotected--;
  pthread_mutex_lock(&lock[1]);
  non_linear = 4.2;
  pthread_mutex_unlock(&lock[1]);
  local_var2 = many_refs;
  return NULL;
}

int main() {
  pthread_t t1, t2, t[2];

  int i;
  for(i = 0; i < 2; i++)
  {
    pthread_mutex_init(&lock[i], NULL);
    pthread_create(&t[i], NULL, my_func, NULL);
  }

  pthread_mutex_init(&lock1, NULL);
  pthread_mutex_init(&lock2, NULL);

  pthread_create(&t1, NULL, my_func, NULL);
  pthread_create(&t2, NULL, my_func, NULL);

  return 1;
}
```

This is Locksmith’s output for this program:

```
Warning: Possible data race: &partly_unprotected:test.c:5 is not protected!
 references:
  dereference of &partly_unprotected:test.c:5 at test.c:17
    &partly_unprotected:test.c:5
  locks acquired:
    concrete lock2:test.c:41
    lock2:test.c:3
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44

  dereference of &partly_unprotected:test.c:5 at test.c:21
    &partly_unprotected:test.c:5
  locks acquired:
    <empty>
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44


Warning: Possible data race: &non_linear:test.c:6 is protected by non-linear or
concrete lock(s):
  *lock:test.c:3
  non linear concrete lock[i]:test.c:36
 references:
  dereference of &non_linear:test.c:6 at test.c:24
    &non_linear:test.c:6
  locks acquired:
    *lock:test.c:3
    non linear concrete lock[i]:test.c:36
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44


Warning: Possible data race: &many_refs:test.c:7 is not protected!
 references:
  dereference of &many_refs:test.c:7 at test.c:14
    &many_refs:test.c:7
  locks acquired:
    concrete lock1:test.c:40
    lock1:test.c:3
```

```
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44

  dereference of &many_refs:test.c:7 at test.c:18
    &many_refs:test.c:7
  locks acquired:
    concrete lock2:test.c:41
    lock2:test.c:3
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44

  dereference of &many_refs:test.c:7 at test.c:20
    &many_refs:test.c:7
  locks acquired:
    <empty>
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44

  dereference of &many_refs:test.c:7 at test.c:26
    &many_refs:test.c:7
  locks acquired:
    <empty>
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44


Warning: Possible data race: &completely_unprotected:test.c:8 is not protected!
 references:
  dereference of &completely_unprotected:test.c:8 at test.c:22
    &completely_unprotected:test.c:8
  locks acquired:
    <empty>
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44
```

As we can see, of all the variables used, many_refs, completely_unprotected and partly_unprotected are not protected and thus prone to data races. In the case of completely unprotected, it is used twice (decrement both reads and writes a value)

and no lock was ever acquired (and released) at this point. The partly_unprotected variable has two references: in the first one it acquired one lock before being accessed but in the second one it didn't do so. In the case of many_refs, it is used in 4 statements but is only protected in two of them. Non-linear is only accessed once but it uses a non-linear lock. As for the other variables, shared_var is accessed once and has got a lock at that point, so it is protected. local_var and local_var2 are thread local variables, so they are not shared with the other thread and there is no need to check if they are protected.

Notice that the warnings are shown in the order their respective variables were first declared. While this is not really problematic in small programs, it is in larger ones. That's because of the way Locksmith works. Since it doesn't actually run the program, we can't know for certain if those warnings are indeed data races or false alarms. So, just in case we have dozens of warnings, it is a good idea to rank the warnings, from the most important to the least. But how do we know which warnings are more important than the others? We need to set a couple of evaluation criteria first.

## Evaluation Criteria

In order to sort the warnings, Locksmith uses certain evaluation criteria. Warnings are sorted in descending order of importance. But what makes a warning important in the first place? For the programmer who checks his program for data races, an important warning would be either something that affects the program in a significant way or something that is more likely to be a data race. This is not always possible to determine if that's the case, so we use some approximations based on the data we can gather from the static analysis. That all that being said, the criteria used in Locksmith are as follows:

**The number of times a memory address is accessed for writing**
As was already mentioned, in order for data races to exist, the threads have to write in a shared memory address. If there are multiple writing operations for a specific variable, it propably means that it would affect the program more and needs more to be done to fix the data race than a race with fewer writing operations.

**The number of total references of a memory address**
Each access of a memory address constitutes a reference to it. Similarly with the first criterion, if a variable is accessed a lot in a program, it probably means that it is more impactful for the program than a variable that is accessed fewer times. In this case, a warning regarding the former variable will be considered more important than the latter.

**Locks acquired by a variable**
Locks prevent simultaneous access to a memory address by multiple threads. This usually prevents data races, however, locks should be acquired every time before a shared variable is accessed and released each time after it is done. It only needs to be accessed once without a lock for a data race to occur. While a data race would be equally disastrous for a program if a variable used no locks and another variable used locks in a few read/write operations, the latter is easier to fix and thus will be considered less important than the former.

**A memory address is protected by non-linear locks**
Sometimes a memory address may be protected by a lock. However, in some cases, different locks may be created in the same line in the program (e.g. inside a loop) and using these locks my cause a warning because the locks would be non-linear. Because Locksmith is unable to tell apart two function calls in the same statement, it cannot possibly know whether it can cause an actual data race or not. As a result, those warnings are not deemed important.

So, how exactly are these criteria implemented inside Locksmith? The next section clears this out.

Implementation

Each shared location has a value called rho_priority_value. The higher that number is, the higher the importance of a warning regarding it. These locations are stored in the all_concrete_rho reference. Throughout Locksmith's analysis, it updates these values whenever necessary with a positive modifier depending on the evaluation criteria that increase the importance of the warnings. For example, a variable that is accessed once for writing (+20000) receives a larger modifier than a variable that is accessed once for reading (+10000). Similarly, it gives a negative modifier depending on the evaluation criteria that decrease the importance of the warnings. For example, a variable that acquired one lock (-10000) receives a smaller negative modifier than a variable with a non-linear lock (-1000000000). Once the analysis is done, it sorts all_concrete_rho in descending order and prints those who may cause warnings. It is important to note that rho_priority_value will be updated for all shared locations during the analysis, regardless if it leads to a race or not.

Now, consider the previous program again. If we run Locksmith again by ranking the warnings (which does by default) we get this:

```
Warning: Possible data race: &many_refs:test.c:7 is not protected!
 references:
  dereference of &many_refs:test.c:7 at test.c:14
    &many_refs:test.c:7
```

```
  locks acquired:
    concrete lock1:test.c:40
    lock1:test.c:3
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44

  dereference of &many_refs:test.c:7 at test.c:18
    &many_refs:test.c:7
  locks acquired:
    concrete lock2:test.c:41
    lock2:test.c:3
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44

  dereference of &many_refs:test.c:7 at test.c:20
    &many_refs:test.c:7
  locks acquired:
    <empty>
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44

  dereference of &many_refs:test.c:7 at test.c:26
    &many_refs:test.c:7
  locks acquired:
    <empty>
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44


Warning: Possible data race: &completely_unprotected:test.c:8 is not protected!
 references:
  dereference of &completely_unprotected:test.c:8 at test.c:22
    &completely_unprotected:test.c:8
  locks acquired:
    <empty>
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44
```

```
Warning: Possible data race: &partly_unprotected:test.c:5 is not protected!
 references:
  dereference of &partly_unprotected:test.c:5 at test.c:17
    &partly_unprotected:test.c:5
  locks acquired:
    concrete lock2:test.c:41
    lock2:test.c:3
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44

  dereference of &partly_unprotected:test.c:5 at test.c:21
    &partly_unprotected:test.c:5
  locks acquired:
    <empty>
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44


Warning: Possible data race: &non_linear:test.c:6 is protected by non-linear or
concrete lock(s):
  *lock:test.c:3
  non linear concrete lock[i]:test.c:36
 references:
  dereference of &non_linear:test.c:6 at test.c:24
    &non_linear:test.c:6
  locks acquired:
    *lock:test.c:3
    non linear concrete lock[i]:test.c:36
  in: main at test.c:30 -> test.c:37
  in: main at test.c:30 -> test.c:43
  in: main at test.c:30 -> test.c:44
```

As we can see, the order of the warnings appears to be completely different. This time, many_refs appears first. In total, it has 2 writing operations, 3 reading operations and acquired locks twice for a value of 2*20000 + 3*10000 + 2*(-10000) = 50000. Next up is completely_unprotected with 1 writing operation and 1 reading operation for a value of 20000 + 10000 = 30000. After that we have partly_unprotected with 1 writing operation, 1 reading operation and 1 lock acquired for a value of 20000 + 10000 – 10000 = 20000. Finally, we have non_linear with 1 writing operation, 1 lock acquired and 1 non-linear lock for a value of 20000 – 10000 – 1000000000 = -999990000. As we can see, the warnings are indeed mentioned in descending order of their rho_priority_values.

## Conclusion

So, to sum up, Locksmith is a program that analyzes other programs written in C in order to find any possible data races. Then, it ranks all potential warnings depending on their potential impact on the program and the likelihood of it being a false positive result. This makes it easier for a programmer to acknowledge and eventually remove the data races.