Dynamic Race Detection for Binary Code

CSCE 689

Project Report

Jianan Liu

226004489

Dynamic Race Detection for Binary Code Algorithm

Jianan Liu 226004489
liujianan@tamu.edu

Abstract – **Multithreaded programs are notoriously prone to race conditions. A lot of static and dynamic data race detection tools have been developed. This project implements binary race detecting tool based on pin by using the FastTrack algorithm. It shows that FT algorithm reduces the complexity.**

Keywords – **Data Race, happensbefore, FastTrack**

Ⅰ. Overview

A race condition occurs when a program's execution contains two accesses to the same memory location that are not ordered by the happens-before relation, where at least one of the accesses is a write.

Multithreaded programs are notoriously prone to race conditions. Since dynamic data race analysis can pinpoint race conditions in large and complex application, for this project I will implement a dynamic data race detection tool for binary code. In general, dynamic race detectors fall into two categories, depending on whether they report false alarms. Precise race detectors never produce false alarms. The problem is that vector clocks are expensive because they record information about each thread in a system. For imprecise race detectors, they may provide better coverage but can report false alarms on race-free programs.

Therefore, this project applies FastTrack algorithm to implement a dynamic data race detection tool for binary code. FastTrack algorithm uses an adaptive representation for the happens-before relation. For almost all operations of the target program, it requires only constant space and supports constant-time operations.

This project is based on Pin tool, a dynamic binary instrumentation framework that enables the creation of dynamic program analysis tools. My original idea is that firstly I can implement normal happens-before algorithm because there are a lot of materials/information we can study from the Internet, after that, I add epoch form to improve happens-before algorithm to FastTrack algorithm. However, because of the logical complexity of implementing FastTrack algorithm, I only successfully implemented a dynamic data race detection tool based on normal happens-before relation. The reason why I failed to implement FastTrack algorithm or the problem I had is that comparing normal happens-before relation using vector clocks, FastTrack algorithm needs another form: epoch to record information, which means we need to write codes to connect C++ memory with not only vector clocks also epoch. When I tried to add epoch form to the code of happens-before algorithm, it's much more difficult than I thought, there are serval bugs. Further debugging works need to be done in the future. But at least by doing this project, I implement happens-before algorithm and have a general idea of how to implement FastTrack algorithm. In the end, I use some simple examples to test my data race detection tool, the result shows that it can detect the race condition.

Ⅱ.Methodology/Approach

1. FastTrack algorithm

The key insight behind FastTrack is that the full generality of vector clocks is not necessary in over 99% of these read and write operations: a more lightweight representation of the happens-before information can be used instead. Only a small fraction of operations performed by the target program necessitate expensive vector clock operations.

For detecting Write-Write Races, we first consider how to efficiently analyze write operations. For normal happens before relation algorithm like DJIT+, it compares the whole vector clocks W_x (from the first write to x) with current corresponding thread's vector clocks to determine whether there is a race. A careful inspection reveals, however, that it is not necessary to record the entire vector clock from the first write to x. Assuming no races have been detected on variable x so far, then all writes to x are totally ordered by the happens-before relation, and so the only critical information that needs to be recorded is the clock and identity of the thread performing the last write. This information is then sufficient to determine if a subsequent write to x is in a race with any preceding write.

Therefore, FastTrack algorithm uses a pair of a clock c and a thread t as an epoch, denoted c@t. Although rather simple, epochs provide the crucial lightweight representation for recording sufficiently-precise aspects of the happens-before relation efficiently. Unlike vector clocks, an epoch requires only constant space, independent of the number of threads in the program, and copying an epoch is a constant-time operation.

An epoch c@t happens before a vector clock V ($c@t \leq V$) if and only if the clock of the epoch is less than or equal to the corresponding clock in the vector.

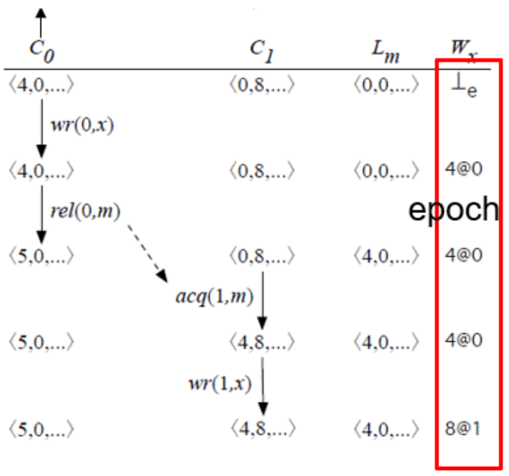
$$c@t \leq V \text{ ifff } c \leq V(t)$$

For detecting Write-Read Races is also straightforward. On each read from x with current vector clock C_t, we check that the read happens after the last write via the same O(1)-time comparison $W_x \leq C_t$.

For detecting Read-Write Race conditions is more difficult. Unlike write operations, which are totally ordered (assuming no race conditions detected so far), reads are not totally ordered even in race-free programs. Thus, a write to a variable x could potentially conflict with the last read of x performed by any other thread, not just the last read in the entire trace seen so far. Hence, we may need to record an entire vector clock R_x, in which $R_x(t)$ records the clock of the last read from x

by thread t. However, according to the paper, we can optimize this by avoiding avoid keeping a complete vector clock in many cases. More discussion can be found in the paper[1].

Now we can use a simple example from the paper[1] to compare the FastTrack and normal happens before (Vector Clocks) and thus show how FastTrack works.



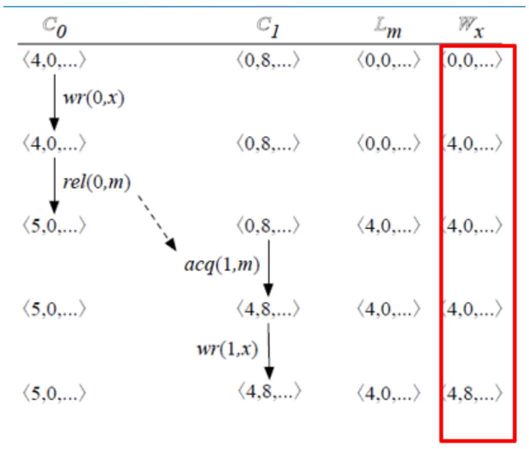


Fig1.(a)FastTrack

Fig1. (b) happens before (vector clock)

Consider the fragment from an execution trace as Fg1. shows, where we include the

relevant
instrumentation state: the vector clocks C_0 and C_1 for threads 0 and 1; and the vector clocks L_m for the last release of lock m. For FastTrack, it uses epoch to record the information of last write to variable x; for happens before (vector clock), it uses vector clock W_x to keep the whole information of write operation to variable. We show two components for each vector clock.

For happens before (vector clock), At the second write, it compares every corresponding component of the vector clocks:

$$W_x = < 4,0 \dots > \in < 4,8 \dots >$$

Which is O(n) time comparison. Since this check passes, the two writes are not concurrent, and no race condition is reported.

For FastTrack algorithm, after using the optimized representation: epoch, FastTrack analyzes same trace using a compact instrumentation state that records only a write epoch W_x for variable x, rather than the entire vector clock W_x, reducing space overhead. At the first write to x, FastTrack performs an O(1)-time epoch write W_x := 4@0 . FASTTRACK subsequently ensures that the second write is not concurrent with the preceding write via the O(1)-time comparison:

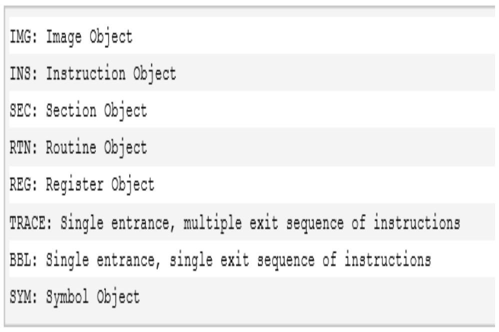
$$W_x = 4@8 \leq < 4,8 \dots > = C_1$$

2. PIN - A Dynamic Binary Instrumentation Tool

Pin is a dynamic binary instrumentation framework that enables the creation of dynamic program analysis tools. Pin allows a tool to insert arbitrary code (written in C or C++) in arbitrary places in the executable. The code is added dynamically while the executable is running. This also makes it possible to attach Pin to an already running process.

Pin provides a rich API that abstracts away the underlying instruction set idiosyncracies and allows context information such as register contents to be passed to the injected code as parameters. Pin automatically saves and restores the registers that are overwritten by the injected code so the application continues to work. Limited access to symbol and debug information is available as well.

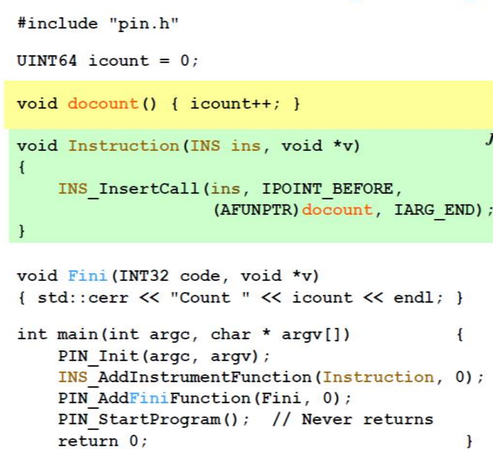
Pin provides callback functions with different granularities. In general, it can be divided into these levels, as the Fig.2 shows.



```
IMG: Image Object
INS: Instruction Object
SEC: Section Object
RTN: Routine Object
REG: Register Object
TRACE: Single entrance, multiple exit sequence of instructions
BBL: Single entrance, single exit sequence of instructions
SYM: Symbol Object
```

Fig. 2

Now we use a simple example in Fig. 3 to show how to use Pin.



```
#include "pin.h"
UINT64 icount = 0;

void docount() { icount++; }

void Instruction(INS ins, void *v)
{
    INS_InsertCall(ins, IPOINT_BEFORE,
                   (AFUNPTR)docount, IARG_END);
}

void Fini(INT32 code, void *v)
{ std::cerr << "Count " << icount << endl; }

int main(int argc, char * argv[])          {
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();  // Never returns
    return 0;                               }
```

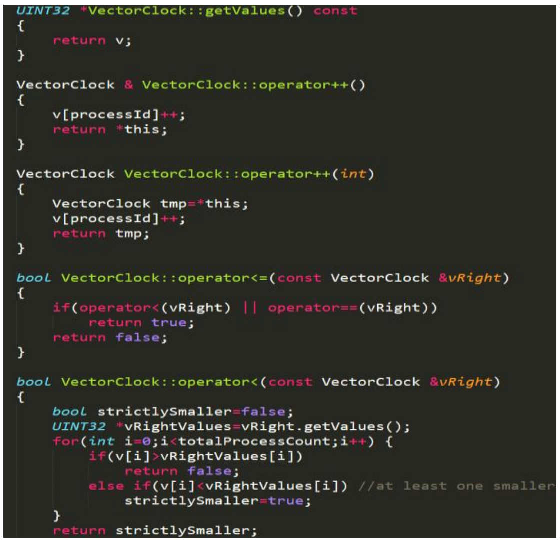
Fig. 3

The yellow part in Fig.3 is our analysis function, it defines what to do when instrumentation is activated. In this example, the do-count function increments the counter. In the green part, the function Instruction is call back function, the parameter IPINT_BEFORE defines where instrumentation is inserted, the parameter do-count is our analysis function, so every time an instruction is executed, our analysis function will be called. Therefore, the logical part we need to write is the yellow part: analysis function.

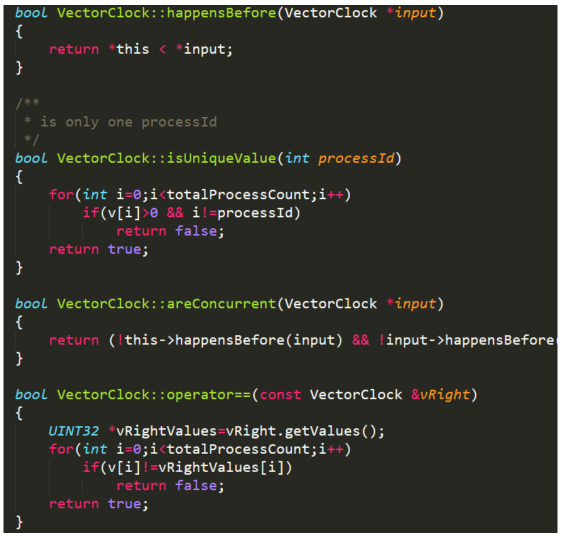
III. Implementation

In this part, I will use some code snipes to show how to implement this detection tool, the first part is how to implement Vector Clock, which is very important. both FastTrack and normal happens before algorithm needs VC to record information. The second part is how to implement happens before relation combined with C++ memory management.

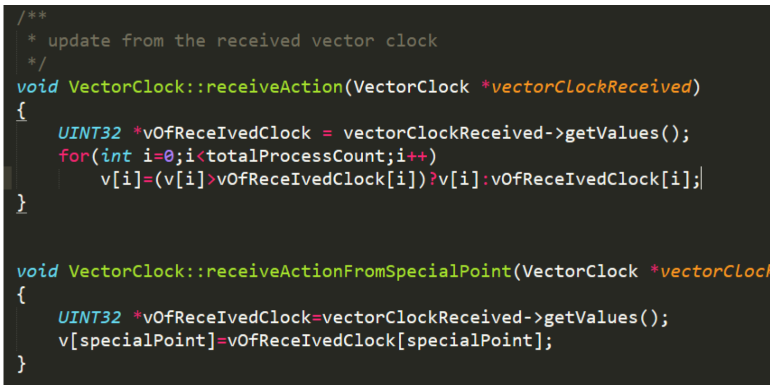
1. Vector Clock



```
UINT32 *VectorClock::getValues() const
{
    return v;
}

VectorClock & VectorClock::operator++()
{
    v[processId]++;
    return *this;
}

VectorClock VectorClock::operator++(int)
{
    VectorClock tmp=*this;
    v[processId]++;
    return tmp;
}

bool VectorClock::operator<=(const VectorClock &vRight)
{
    if(operator<(vRight) || operator==(vRight))
        return true;
    return false;
}

bool VectorClock::operator<(const VectorClock &vRight)
{
    bool strictlySmaller=false;
    UINT32 *vRightValues=vRight.getValues();
    for(int i=0;i<totalProcessCount;i++) {
        if(v[i]>vRightValues[i])
            return false;
        else if(v[i]<vRightValues[i]) //at least one smaller
            strictlySmaller=true;
    }
    return strictlySmaller;
```



```
bool VectorClock::happensBefore(VectorClock *input)
{
    return *this < *input;
}

/**
 * is only one processId
 */
bool VectorClock::isUniqueValue(int processId)
{
    for(int i=0;i<totalProcessCount;i++)
        if(v[i]>0 && i!=processId)
            return false;
    return true;
}

bool VectorClock::areConcurrent(VectorClock *input)
{
    return (!this->happensBefore(input) && !input->happensBefore
}

bool VectorClock::operator==(const VectorClock &vRight)
{
    UINT32 *vRightValues=vRight.getValues();
    for(int i=0;i<totalProcessCount;i++)
        if(v[i]!=vRightValues[i])
            return false;
    return true;
}
```

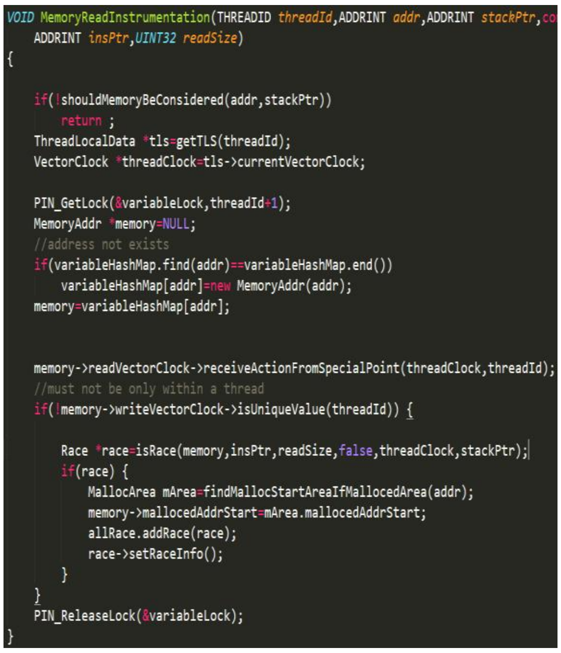


```
/**
 * update from the received vector clock
 */
void VectorClock::receiveAction(VectorClock *vectorClockReceived)
{
    UINT32 *vOfReceIvedClock = vectorClockReceived->getValues();
    for(int i=0;i<totalProcessCount;i++)
        v[i]=(v[i]>vOfReceIvedClock[i])?v[i]:vOfReceIvedClock[i];
}


void VectorClock::receiveActionFromSpecialPoint(VectorClock *vectorCloc
{
    UINT32 *vOfReceIvedClock=vectorClockReceived->getValues();
    v[specialPoint]=vOfReceIvedClock[specialPoint];
}
```

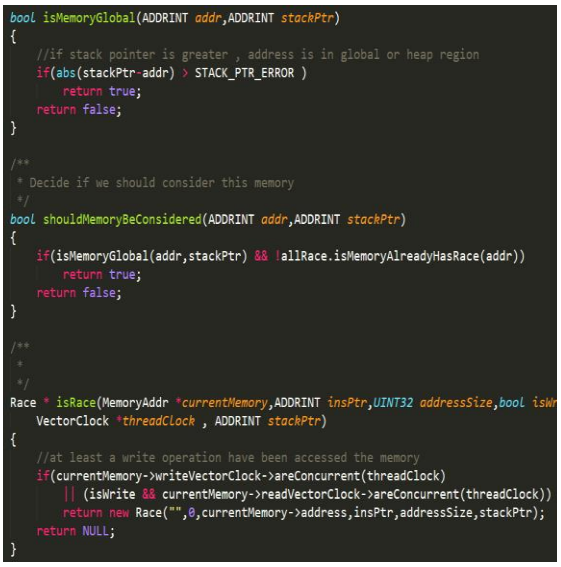
Fig.4 (a) (b) (c)

Fig.4 (a) (b) (c) show the main functions of vector clock (vector clock is a class we construct). From Fig.4 (a) (b) (c), we can see that there are some functions like overloaded operators <= ,<, ++… which are very important. The reason is that for happens before relation algorithm, it's very important to check and compare different vector clocks to see which component happens before the other. Also, in the Fig. 4 there are two functions: receiveAction and receiveActionFromSpecialPoint. Their aim is to update the current vector clocks from the receive vector clock to make sure the updating mechanism work well. The function isUniqueValue is to check whether operation happens in the same thread, if return true, we do not need to check race condition. These functions: happensBefore, areConcurrent help us to realize whether there are 2 operations working concurrently. Therefore, by writing these functions, we can get the functions of vector clock that we need.

2. Check race condition



```
VOID MemoryReadInstrumentation(THREADID threadId,ADDRINT addr,ADDRINT stackPtr,co
    ADDRINT insPtr,UINT32 readSize)
{

    if(!shouldMemoryBeConsidered(addr,stackPtr))
        return ;
    ThreadLocalData *tls=getTLS(threadId);
    VectorClock *threadClock=tls->currentVectorClock;

    PIN_GetLock(&variableLock,threadId+1);
    MemoryAddr *memory=NULL;
    //address not exists
    if(variableHashMap.find(addr)==variableHashMap.end())
        variableHashMap[addr]=new MemoryAddr(addr);
    memory=variableHashMap[addr];


    memory->readVectorClock->receiveActionFromSpecialPoint(threadClock,threadId);
    //must not be only within a thread
    if(!memory->writeVectorClock->isUniqueValue(threadId)) {

        Race *race=isRace(memory,insPtr,readSize,false,threadClock,stackPtr);
        if(race) {
            MallocArea mArea=findMallocStartAreaIfMallocedArea(addr);
            memory->mallocedAddrStart=mArea.mallocedAddrStart;
            allRace.addRace(race);
            race->setRaceInfo();
        }
    }
    PIN_ReleaseLock(&variableLock);
}
```



```
bool isMemoryGlobal(ADDRINT addr,ADDRINT stackPtr)
{
    //if stack pointer is greater , address is in global or heap region
    if(abs(stackPtr-addr) > STACK_PTR_ERROR )
        return true;
    return false;
}

/**
 * Decide if we should consider this memory
 */
bool shouldMemoryBeConsidered(ADDRINT addr,ADDRINT stackPtr)
{
    if(isMemoryGlobal(addr,stackPtr) && !allRace.isMemoryAlreadyHasRace(addr))
        return true;
    return false;
}

/**
 *
 */
Race * isRace(MemoryAddr *currentMemory,ADDRINT insPtr,UINT32 addressSize,bool isWr
    VectorClock *threadClock , ADDRINT stackPtr)
{
    //at least a write operation have been accessed the memory
    if(currentMemory->writeVectorClock->areConcurrent(threadClock)
        || (isWrite && currentMemory->readVectorClock->areConcurrent(threadClock))
        return new Race("",0,currentMemory->address,insPtr,addressSize,stackPtr);
    return NULL;
}
```

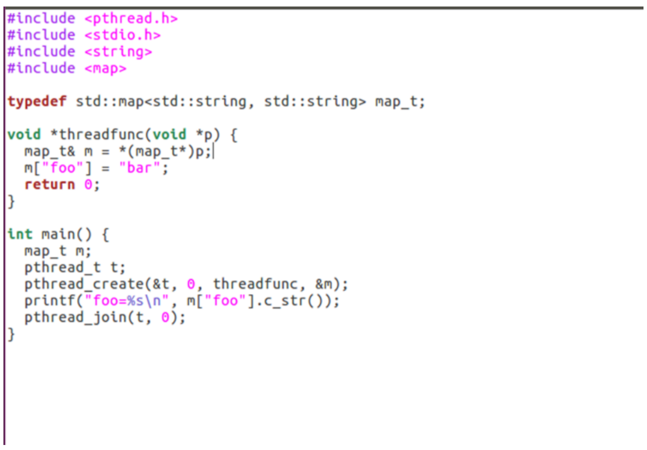
Fig. 5 (a) (b)

Fig.5 (a) shows when a read operation happens, we need to check whether there is a write-read race condition. The first step is that we need to use another function called: shouldMemoryBeConsidered to check whether the memory address is in global or the memory already has race condition. If the memory is in global and hasn't been checked before, we need to check. Now read operation happens to this memory, we need to update its readVectorClock. And we also need to check if other operations (write operations) works in the same thread. If it's not, we now check whether it has race condition in this memory. We use isRace function to detect it by function areConcurrent of corresponding vector clocks. And if we detect there is a race condition, we return new race, a class we construct which includes the relative information of this race condition. Fig. 5(b) shows the details of shouldMemoryBeConsidered, isRace functions. When a write operation happens, we need to check write-write race condition, read-write race condition. The implementation detail is very similar to checking whether there is a write-read race condition when a read operation happens. The code snipe showing is normal happens before relation algorithm, for the FastTrack algorithm, we need to construct a new epoch class whose functions are similar to the Vector Clock class, and use epoch to replace some vector clock. Because of limitation of time and complexity of logics, I haven's successfully implemented this part, I hope further work must be done in the future.

These above are two parts I want to highlight because they show how happens before relation works. For other codes, I will not discuss in the report.

Ⅳ. Experimental Results

Now we can implement this tool on a simple testing program, this testing program is shown in the Fig. 6

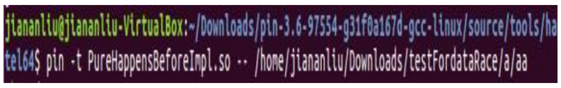


```
#include <pthread.h>
#include <stdio.h>
#include <string>
#include <map>

typedef std::map<std::string, std::string> map_t;

void *threadfunc(void *p) {
  map_t& m = *(map_t*)p;
  m["foo"] = "bar";
  return 0;
}

int main() {
  map_t m;
  pthread_t t;
  pthread_create(&t, 0, threadfunc, &m);
  printf("foo=%s\n", m["foo"].c_str());
  pthread_join(t, 0);
}
```

Fig. 6

By looking at this code, we can find that both main thread and t thread try to access to the map m, t tread uses threadfunc to assign value “bar” to key “foo” (write operation), and at the same time main thread wants to printf the value form the key “foo”, therefore the race condition happens.

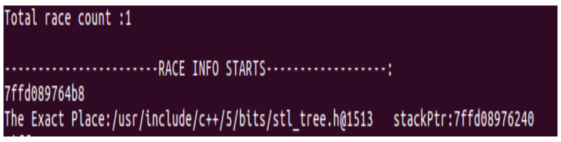
Since our tool is dynamic tool, which means that in fact we do not need to know our original code. Therefore firstly we compile our testing program to make it executable file. Then we use the command: “pin - t <our tool path and name > -- <testing executable file path and name>” to detect, which is shown in Fig. 7. “aa” is our testing executable file name.



```
jiananliu@jiananliu-VirtualBox:~/Downloads/pin-3.6-97554-g31f0a167d-gcc-linux/source/tools/ha
tel64$ pin -t PureHappensBeforeImpl.so -- /home/jiananliu/Downloads/testFordataRace/a/aa
```

Fig.

The result is shown in Fig. 8 below:



```
Total race count :1


---------------------RACE INFO STARTS-----------------:
7ffd089764b8
The Exact Place:/usr/include/c++/5/bits/stl_tree.h@1513   stackPtr:7ffd08976240
```

Fig. 8

We can see that the tool detects the race condition and according to the race information, the race condition happens in the memory address: 7ffd089764b8.

I also use another testing program to test this tool, but it does not successfully test the race condition. Since happen before relation tool is a precise race detector, which never produces false alarms. I guess sometimes it will miss the race condition. More tests need to be done in the future to see if this tool can work practically.

V. Conclusion and Further work

This project compares the traditional happens before relation algorithm with FastTrack algorithm showing that FastTrack algorithm reduces the time and space complexity. Also, in this project I uses pin as framework to implement a dynamic binary race detecting tool and the tool can successfully detect some race conditions from some testing programs. There are two further works need to be done: firstly, I still need to successfully implement FastTrack algorithm, which I have ideas but need more time; Secondly, more tests need to be done to see if my tool can work practically.

ACKNOWLEDGMENT

I would like to thank Professor Jeff Huang for teaching me “Fundamentals of Software Analysis” course. By studying this course, I learn how to detect the program from several ways, which gives me a new perceptive to see the code. and for this project, it has helped me get a deeper understanding of Software Analysis, Race Detection, Happens before relation, FastTrack algorithm, I also need to thank my classmates, by listening to their perspectives and answers, I gain a lot of knowledge about how to think and how to code. Overall, it’s a challenging course and I learn a lot from this course.

REFERENCES

[1] Flanagan, Cormac, and Stephen N. Freund. "FastTrack: efficient and precise dynamic race detection." *ACM Sigplan Notices*. Vol. 44. No. 6. ACM, 2009.

[2] Michel Raynal, Mukesh Singhal. "Capturing Causality in Distributed Systems."

[3] Pin tool tutorial website: https://software.intel.com/sites/landingpage/pintool/docs/97554/Pin/html/index.html