A Comprehensive Security Approach on Data Race Detection and Deepfake Defense

Yu Guo
Western Michigan University, lei8845@gmail.com

Follow this and additional works at: https://scholarworks.wmich.edu/dissertations

Part of the Information Security Commons

Recommended Citation
https://scholarworks.wmich.edu/dissertations/3649
A COMPREHENSIVE SECURITY APPROACH ON DATA RACE DETECTION AND DEEPFAKE DEFENSE

Yu Guo, Ph.D.
Western Michigan University, 2020

In the past decades, even the computer techniques have a significant improvement. The topic about security will be never out of date.

To meet with the requirement of computation, the number of CPU cores has changed from single core to multi-cores. At the same time, the multi-thread programs are also proposed to maximize the advantages of multi-core computing power. While even the performance has been improved, but it also brings some new issues which were never happened on sequential programs called current bugs. Data race is a major type of current bugs, it is happened when multiple threads access the same memory location, and at least one of them is write operation. Compare to general bugs, to detect data race is more difficult and more expensive.

Additionally, the deep learning is another hot topic area in recent years. With the improving of GPU’s performance, the neural network was deployed on GPU rather than CPU, because compare to CPU, GPU has a better computational ability on neural network. Deepfake is a new type of technique which was created based on deep learning. It is a means to swap faces realistically with a low cost in a short time. Because it is fake and a production of deep learning, so called “Deepfake”. This technique could be widely used on education, art, and entertainment area. However, it is also found in generating revenge porn, fake news, economic fraud. Because of its realistic characteristic, it is very hard to
distinguish the authenticity of a picture or a video attacked by Deepfake.

A lot of detectors have been released on both races’ detection and Deepfake’s detection areas. For data races, the static analyze will have a lot false positive, while the dynamic analyze has fewer, but it will bring a huge extra overhead. To reduce the cost of dynamic analyze, the sampling strategy has been introduced, but current sampling tool reduced the overhead based on reducing the accuracy. As for the Deepfake part, some detection tools distinguish the fake materials by detecting abnormal biological information or the technical defects which have been fixed by the newer version of Deepfake strategy. The other branch is to find the consistent between frames, but this kind of method cannot be used to detect fake images. Besides, all the Deepfake detectors cannot guarantee 100% accuracy. Even with the evolution of the Deepfake, the accuracy may become lower and lower.

To address above issue, Atexrace has been presented to detect race which has the low overhead as the state-of-art sampling method and a better accuracy as the detector without sampling applied. Besides, invisible watermark embedding method to defend Deepfake attacking was proposed in this dissertation which has a 100% defense accuracy and never be out of date. And experiments result also confirmed the conclusion.
ACKNOWLEDGEMENTS

The dissertation is the last step to my Ph.D. degree. From 2013, I have searched it for almost 7 years. During this process, I have obtained a lot of helps and supports from my family, professors, and friends. Without them I cannot reach to this step.

Firstly, I would like to express my deepest appreciation to my parents for their support without caring any return. Especially when I encounter difficulties, it is their unconditional support that helps me get through the difficulties.

Secondly, I want to thank my advisor Prof. Zijiang Yang for his patient and support. It is his help to lead me to open the gate of research and make me grow from an ignorant youth to an independent researcher. I also want to thank my committee members: Dr. Li Yang and Dr. Yan Cai. Dr. Li Yang has given me a lot of helps and convenience during my adamic career in Western Michigan University. And without the help of Dr. Yan Cai, I would not publish my first paper.

Thirdly, I sincerely thank the chair of Computer Science department, Dr. Steve Carr for supporting me as a graduate assistant and administrative assistant Ms. Sheryl Todd’s helps during the process.

Finally, I thank all the people who supported me during my Ph.D. career.

Yu Guo
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ........................................................................................................................................ ii
LIST OF TABLES ................................................................................................................................................ vi
LIST OF FIGURES ........................................................................................................................................ vii
CHAPTER 1 .................................................................................................................................................... 1
  1.1 Computer Security ............................................................................................................................... 1
  1.2 Multi-thread Program ......................................................................................................................... 1
  1.3 Process and Thread ............................................................................................................................. 3
  1.4 Data Race ........................................................................................................................................ 5
    1.4.1 Challenge of Race Detection ...................................................................................................... 5
    1.4.2 Current Race Detection Approach ............................................................................................ 7
  1.5 Artificial Intelligent, Machine Learning and Deep Learning ......................................................... 7
    1.5.1 Artificial Intelligent ..................................................................................................................... 7
    1.5.2 Machine Learning ....................................................................................................................... 8
    1.5.3 Deep Learning ........................................................................................................................... 9
    1.5.4 Relationship of Artificial Intelligent, Machine Learning and Deep Learning ..................... 10
  1.6 Neural Networks .............................................................................................................................. 11
  1.7 Deepfake .......................................................................................................................................... 14
  1.8 Current Approach ............................................................................................................................ 15
  1.9 Dissertation’s Contributions and Organization ............................................................................... 15
References ................................................................................................................................................... 17

CHAPTER 2 .................................................................................................................................................. 18
THE ACROSS THREAD AND EXECUTION SAMPLING FOR IN-HOUSE RACE DETECTIONS: ATEXRACE AND ATEXRACEPLUS ............................................................................................... 18
  2.1 Summary ......................................................................................................................................... 18
  2.2 Introduction ..................................................................................................................................... 19
  2.3 Background .................................................................................................................................... 25
    2.3.1 Multi-threaded Programs .......................................................................................................... 25
    2.3.2 Data Race ................................................................................................................................ 25
  2.4 Motivation ....................................................................................................................................... 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.1</td>
<td>Dynamic Data Race Detection</td>
<td>27</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Sampling on Dynamic Data Race Detection</td>
<td>29</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Limitations of Existing Sampling Approaches</td>
<td>31</td>
</tr>
<tr>
<td>2.5</td>
<td>Our Approach</td>
<td>33</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Goal and Challenge</td>
<td>33</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Basic AtexRace Algorithm</td>
<td>35</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Limitations of Basic AtexRace</td>
<td>37</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Optimizations</td>
<td>40</td>
</tr>
<tr>
<td>2.5.5</td>
<td>AtexRacePlus Algorithm</td>
<td>44</td>
</tr>
<tr>
<td>2.5.6</td>
<td>AtexRacePlus on Example Program</td>
<td>52</td>
</tr>
<tr>
<td>2.5.7</td>
<td>Discussion on AtexRacePlus</td>
<td>53</td>
</tr>
<tr>
<td>2.6</td>
<td>Evaluation</td>
<td>56</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Implementation</td>
<td>57</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Benchmarks</td>
<td>58</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Experiment Setup</td>
<td>59</td>
</tr>
<tr>
<td>2.6.4</td>
<td>Evaluation of Efficiency</td>
<td>59</td>
</tr>
<tr>
<td>2.6.5</td>
<td>Result Analysis on MySQL</td>
<td>64</td>
</tr>
<tr>
<td>2.6.6</td>
<td>Evaluation of Accuracy</td>
<td>66</td>
</tr>
<tr>
<td>2.7</td>
<td>Discussion</td>
<td>67</td>
</tr>
<tr>
<td>2.8</td>
<td>Related Works</td>
<td>68</td>
</tr>
<tr>
<td>2.9</td>
<td>Conclusion</td>
<td>70</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>72</td>
</tr>
</tbody>
</table>

CHAPTER 3

DEEP DEFENCE BY EMBEDDING INVISIBLE WATERMARK | 77

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Summary</td>
<td>77</td>
</tr>
<tr>
<td>3.2</td>
<td>Introduction</td>
<td>78</td>
</tr>
<tr>
<td>3.3</td>
<td>Background</td>
<td>82</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Deepfakes</td>
<td>82</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Videos and Pictures Color Space</td>
<td>83</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Discrete Cosine Transform</td>
<td>85</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Quantization Index Modulation (QIM)</td>
<td>86</td>
</tr>
<tr>
<td>3.4</td>
<td>Motivations</td>
<td>88</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Beneficial Use and Harmful Use of Deepfakes</td>
<td>88</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Limitations of Existing Approaches</td>
<td>88</td>
</tr>
<tr>
<td>3.5</td>
<td>Our Approach</td>
<td>89</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Goal and Challenges</td>
<td>89</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Process of Approach</td>
<td>89</td>
</tr>
</tbody>
</table>
LIST OF TABLES

1.1 Possible with the threads and statements increasing ................................................................. 6

2.1 Sample rate of each function pairs in the program of Figure 2 .................................................. 53

2.2 Time cost of seven tools on Parsec ............................................................................................ 60

2.3 Overhead and number of unique data race in Parsec ............................................................... 62

3.1 Test videos’ details .................................................................................................................. 96
LIST OF FIGURES

1.1 Microprocessor trend data ........................................................................................................ 2
1.2 Comparison between sequential and concurrent programs ..................................................... 3
1.3 Data race example .................................................................................................................. 5
1.4 Challenge of detecting concurrency bugs ............................................................................... 6
1.5 Relationship of AI, machine learning and deep learning ..................................................... 10
1.6 Neural network model ........................................................................................................... 12
1.7 Neuron model ....................................................................................................................... 12
1.8 Scalable neuron model ......................................................................................................... 13
2.1 A code sketch with two threads and four function calls. .................................................... 21
2.2 A program with races on variable x between line 8 and line 29 ............................................ 26
2.3 An illustration on the instrumentation and race detection for each memory access ............. 28
2.4 Four executions scenarios of the program in Figure 2 ......................................................... 30
2.5 A program consisting of two threads with two data races on variables x (lines 6 and 23) and y (lines 19 and 10). .......................................................... 39
2.6 The overview of AtexRacePlus framework ......................................................................... 46
2.7 Ideal overhead changes with increasing executions. ............................................................ 54
2.8 The linear regression of overhead quotient for different sampling techniques with increasing executions on Parsec benchmark applications .................................................. 63
2.9 Linear regression of overhead quotient with increasing executions on MySQL ............... 65
2.10 The cumulative number of function pairs ......................................................................... 66
2.11 The number of races detected by five techniques on MySQL .......................................... 67
3.1 Deepfake attacking ................................................................................................................. 79
3.2 Training process ....................................................................................................................... 83
3.3 Repair process ........................................................................................................................ 83
3.4 Quantization Index Modulation (QIM) ................................................................................... 86
3.5 68 facial landmarks layout ..................................................................................................... 90
3.6 Embedding process ................................................................................................................ 91
3.7 Extracting process .................................................................................................................. 94
3.8 Comparison original and embedded watermark ........................................................................ 97
3.9 SSIM ..................................................................................................................................... 99
3.10 PSNR .................................................................................................................................. 100
3.11 NCC ................................................................................................................................... 101
3.12 Watermark extraction comparison ...................................................................................... 102
3.13 Watermark robust test result ............................................................................................... 103
CHAPTER 1
INTRODUCTION

1.1 Computer Security

In the past decades, even the computer techniques have a significant improvement. The topic about security will never out of date.

Computer security is a very important area in computer science. It protects the information system from intentional or unintentional destruction. Intentional destruction includes through attacking the vulnerabilities and defects of information system or by high tech to destroy, change and steal to achieve the malicious goals, such as blackmail, bilk, reprisal, etc. While the unintentional destruction will be caused by the vulnerabilities and defects of information system without external attack, the millennium bug is a very famous bug cause by the defects of the system.

Based on the research of CSIS [1], only the computer security problem will bring $300 billion to $1 trillion loss which is the 0.4% to 1.4% of global GDP.

1.2 Multi-thread Program

To meet with the requirement of computation, the number of CPU cores has changed from single core to multi-cores. Figure 1.1 [2] show the last 40 years of microprocessor tend data. Just as it shows, all indicators increase linearly except the number of logical cores. Multi-cores of CPU only appeared around 2005. At the same time, the multi-thread programs also been created to maximize the advantages of multi-core computing ability.
Unlike the traditional program executed on single core CPU, the multi-thread program could execute the multi-thread concurrently. Compare to sequential execution, current execution could handle more tasks at the same time, in another words, it improves the efficiency and the resource utilization. Figure 1.2 shows the differences between single thread program and multi-thread program. It is obvious that a greater number of threads will bring higher efficiency.

However, the multi-thread program increased efficiency, but it brought new problems which were never happened in single-thread program, i.e. data race, atomicity violation and deadlock. And we will talk about it later.
1.3 Process and Thread

**Process**, a process is a running activity of a program with an independent function on a data set and an independent unit for resource allocation and scheduling by the system. In other word, a process is abstracted by the CPU when a program is running, which means a process a program’s execution is abstracted as a process.

**Thread**, a thread is an entity of process and the basic unit of CPU scheduling and dispatch. It is a smaller independent running unit than process. Also, threads do not own system resources except some essential resources in operation, such as program counter, register and stack. But a thread could share all the resources with other threads which belong to the same process. One thread can create and terminate another thread; multiple threads in the same process can execute concurrently.

**Difference between process and thread**. The main difference between process and thread is they are difference ways of operating system resource management strategy. Processes have independent memory address spaces. After a process crashes, it will not affect other processes in
protected mode, while threads are just different execution paths in a process. Threads have their own
stack and local variables, but there is no separate memory address space between threads. A thread
crashed is equal to the entire process crashed, so a multi-process program is more robust than a multi-
thread program. But when the process is switched, compare to the thread switching, it consumes more
resources and the efficiency is less. Additionally, for some concurrent operations that require
simultaneous and shared certain variables, only multi-thread can be used, not multi-processes. Below
are the main different points between process and thread:

- A program has at least one process, and a process has at least one thread.
- The division scale of threads is smaller than that of processes, which makes the concurrency
  of multi-thread programs high.
- The processes have independent memory unit during execution, and multiple threads share
  the memory, which greatly improves the efficiency of the program.
- Threads are still different from processes during execution. Each independent process has an
  entry for program execution, a sequence of sequential execution, and an exit for program.
  But threads cannot be executed independently, it must be stored in the programs, and the
  programs provide multiple threads for execution control.
- From a logical point of view, the significance of multithreading is that in an application,
  multiple execution parts can be executed simultaneously. However, the operating system
does not regard multiple threads as multiple independent applications to achieve process
  scheduling and management and resource allocation.

**Pros and cons**, thread execution overhead is small, but it is not conducive to resource
management and protection, and the process execution is reversed. Besides, threads are suitable for
running on symmetric multiprocessing machines, while processes can be migrated across machines.

1.4 Data Race

Multi-thread program improves the efficiency and resource utilization. It also brings new issues which were never happened in single-thread program, we called these kind of issues as “concurrency bugs”. Data race is a major type of concurrency bug [3]. Data race will happen when more than one threads access to the same memory location and at least one operation is writing operation [4]. Figure 1.3 shows an example of data race, Functions $f_1$ and $f_2$ are repeatedly executed in thread $t_1$, and $f_3$ and $f_4$ are repeatedly executed in $t_2$. Races occur when $f_1$ and $f_4$ execute simultaneously, and when $f_2$ and $f_3$ execute simultaneously.

```
Thread $t_1$
1. for (...){
2. if(...) $f_1$();
3. else $f_2$();
4. }
5. for (...){
6. if(...) $f_3$();
7. else $f_4$();
8. }
```

Figure 1.3 Data race example

1.4.1 Challenge of Race Detection

Figure 1.4 shows the difference between sequential bug and concurrency bug. Due to the sequential execution order is linear, it is very simple to replay the bugs. As to the concurrency bugs, every execution’s interleaving may different, just as showed, the first execution’s serialized sequence
may ABCDEF, while the next execution serialized sequence may change to BACEDF. Bugs may occur only with specific interleaving. Because the scheduling cannot be controlled, if a bug appears in current execution, and it may disappear in next execution, which brings us a huge trouble to detect it.

![Figure 1.4 Challenge of detecting concurrency bugs](image)

Table 1.1 Possible with the threads and statements increasing

<table>
<thead>
<tr>
<th>Threads numbers</th>
<th>Statements numbers/thread</th>
<th>Number of possible paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1680</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>630630000</td>
</tr>
<tr>
<td>n</td>
<td>m</td>
<td>...</td>
</tr>
</tbody>
</table>

It seems like if we could monitor all the possible serialized sequences, the issues will be solved. This idea is correct, but it is unrealistic. Since the number of the possible paths could expand to an astronomical number, the cost of monitoring all the possible paths is unaffordable and not worth it.
Table 1.1 shows the relationship between the number of possible paths and the number of statements and threads. Only 4 threads and 4 statements, the number of possible paths already reach to 630,630,000. Not to mention the number of threads and statements of applications in real life is much larger than this.

1.4.2 Current Race Detection Approach

Race detection has two branches, one is static analyzes and the other one is dynamic analyzes. The former one has fewer overhead but will report many false positive. Since it is very time-consuming to filter false positives [5][6][7][8], it is generally not acceptable in practice. While the dynamic strategy has the opposite performance. Even it could report fewer false positive, but the huge overhead still needs to take care of. Most of the race detector will bring 400%-800% [9][10], which is an unacceptable number.

From last section, we know that it is impossible to detect all the races in the concurrent programs. Most of the race detectors just detect as much as the run time data race, which means each test report is different. Since it is not possible to monitor all the races, to reduce the overhead, the sampling strategy has been introduced. But the current sampling race detectors are still not perfect since they missed too much races.

1.5 Artificial Intelligent, Machine Learning and Deep Learning

1.5.1 Artificial Intelligent

Not as the multi-thread program, Artificial Intelligent is a concept that has been around for a long
time, but it has only been vigorously developed in recent years. In 1956, the concept of "artificial intelligence" was proposed at Dartmouth conference, aimed at using the computer to construct complex machines with the same essential characteristics as human intelligence. After that, artificial intelligence techniques just stay in the experimental stage in research lab. In the following decades, people's attitude towards artificial intelligence is polarized. Some people believe that artificial intelligence is the future of human civilization, while the other people consist research on it is meaningless and should be abandoned. The debate continued until 2012.

With the increase in data volume, the improvement of computing power and the emergence of new machine learning algorithms (deep learning), artificial intelligence began to explode. Artificial intelligence is usually divided into weak artificial intelligence and strong artificial intelligence. The former equips the machine with the ability to observe and perceive, and can do some understanding and inference to certain extent, while the strong artificial intelligence allows the machine to acquire adaptive capabilities to solve some problems that have not been encountered before. However, the current research work is focused on the weak artificial intelligence part, and major breakthroughs are made every year. Most of the artificial intelligence in the film is depicting strong artificial intelligence, which is difficult to achieve in the current real world.

How can weak artificial intelligence achieve breakthroughs, and where does "intelligence" come from? This is mainly due to a new implementation method of artificial intelligence-machine learning.

1.5.2 Machine Learning

The most basic approach of machine learning is to use algorithms to parse data, learn from it, and
then make decisions and predictions about events in the real world. Unlike traditional hard-coded software programs that solve specific tasks, machine learning uses large amounts of data to "train" and learn how to complete tasks from the data through various algorithms.

For example, when we browse the online store, there will often be information about product recommendations. This is the website based on your previous shopping records and the list of favorites to identify which of these are the products you are really interested in and willing to buy. Such a decision model can help the merchant provide recommendations to customers and encourage product consumption.

Machine learning comes from the early artificial intelligence field. Traditional algorithms include decision trees, clustering, Bayesian classification, support vector machines, EM, Adaptive Boost (AdaBoost), and so on. In terms of learning methods, machine learning algorithms can be divided into supervised learning (such as classification problems), unsupervised learning (such as clustering problems), semi-supervised learning, integrated learning, deep learning, and reinforcement learning.

The application of traditional machine learning algorithms in the fields of fingerprint recognition, face detection based on Haar-like feature, and object detection based on Histogram of oriented gradients (HoG) features has basically met the requirements of commercialization or the commercialization of specific scenes, but each step is extremely difficult until the emergence of deep learning algorithms.

1.5.3 Deep Learning

Deep learning is not originally an independent learning method, but it also uses supervised and
unsupervised learning methods to train deep neural networks. However, due to the rapid development of this field in recent years, some unique learning methods have been proposed (such as residual network), so more and more people regard it alone as a learning method.

The initial deep learning is a learning process that uses deep neural networks to solve feature expressions. Deep neural network is not a brand-new concept, it can be roughly understood as a neural network structure with multiple hidden layers. To improve the training effect of deep neural networks, people make corresponding adjustments to the connection methods and activation functions of neurons. In fact, there were many ideas in the early years, but due to insufficient training data and backward computing power, the effect was not satisfactory.

1.5.4 Relationship of Artificial Intelligent, Machine Learning and Deep Learning

Machine learning is a method to realize artificial intelligence, and deep learning is a technology to realize machine learning. Figure 1.5 Visually show the relationship between them.

![Figure 1.5 Relationship of AI, machine learning and deep learning](image)
1.6 Neural Networks

Figure 1.6 shows a simple model of a neural network model. It is a neural network with three layers: input layer, middle layer (hidden layer) and output layer. The circles stand for neurons, we can call them as nodes. In this example there are 4 neurons in input layer, 5 neurons in hidden layer and 3 neurons in output layer. Generally, the number of nodes in the input layer and output layer is often fixed, and the middle layer can be freely given. The arrows in the figure represent the flow of data during the prediction process, and there are different from the data flow during training process. There are two major features in a neural network: neurons and connections. The key features in the structure diagram is not circles (representing "neurons") but connecting lines (representing connections between "neurons"). Each connecting line corresponds to a different weight, which needs to be obtained by training.

The prototype of neuron model was first introduced by McCulloch and Pitts in 1943 which also known as M-P model. The neuron model is a model that includes input, output, and calculation functions. Figure 1.7 is a typical neuron model includes three inputs, one output and 2 calculation functions. Also, each connection has a weight. A neural network training algorithm is to adjust the weight value to the best, so that the prediction effect of the entire network is the best.
We use $i$ to denote an input and $w$ to denote weights. A directed arrow indicating the connection can be understood as follows: at the left side, the transmitted signal is still $i$, and there is a weighting parameter $w$ in the middle. The signal after this weighting will become $i \times w$, so at the right side of the connection, the signal the size becomes $i \times w$. So, the Figure 1.7 could be represented by the following equation:
\[ output = (Sgn) \times (input1 \times w1 + input2 \times w2 + input3 \times w3) \]

Next step is to do some transmission on figure 1.7. We can combine the sum and sng operations into one circle to stand the neuron internal computation. And the output will be split into several output lines which have the same value for the following network. Then we obtained a scalable neuron model, showed as Figure 1.8. Neuron can be regarded as a calculation and storage unit. Computation is the function of neurons to calculate their input. The storage is that the neurons will temporarily store the calculation result and pass it to the next layer.

![Figure 1.8 Scalable neuron model](image)

Compare with current neuron model, the weights in M-P model are pre-set, which means the M-P model has no learning ability. In 1949 Hebb proposed Hebb learning theory [11]. In the theory Hebb thought the weights of the connections can be changed. So, scientists began to consider adjusting the weights to make machine learning. Limited by the computing power at the time, it was until 1960,
Rosenblatt proposed the first single layer neural network, which he called it as “Perceptron” [12]. But Minsky proved the weakness of the perceptron with mathematics, especially the simple classification task like XOR. If the calculation layer is increased to two layers, the amount of calculation is too large, and there is no effective learning algorithm [13]. Until 1986, Rumelhar, Hinton and others proposed the backpropagation (BP) algorithm [14], which solved the complex calculation problem required by the two-layer neural network, thereby driving to use two-layer neural network research works. Based on the two-layer neural network, Hinton proposed multilayer neural network in 2016, thus laying the foundation for deep learning.

Multi-layer neural network is adding extra layer based on the Figure 1.6. So, the original output layer changed to the middle layer, and the new added layer could be the output layer. In this way, we can continue to add more layers to obtain more complexity neural networks. In multi-layer neural networks, the output is also calculated layer by layer. Starting from the outermost layer, after calculating the values of all cells, continue to calculate a deeper layer. Only after the values of all cells in the current layer have been calculated will the next layer be counted. This process is called "forward propagation".

1.7 Deepfake

Deepfake is an artificial intelligence-based image synthesis technology. Since it is the product of deep learning and fake, so called “Deepfake”. Deepfake is used to combine and superimpose existing images and videos onto the source image or video using machine learning techniques called "Generating Adversarial Networks" (GAN). The combination of the existing video and the source
video produces a fake video that shows one or more people performing actions in an event that never happened. Because of its lifelike characteristic, it can hardly tell the truth from the human eye. So, Deepfake may be used to create fake celebrity porn videos or revenge pornography. And it can also be used to produce fake news and malicious pranks. The detection of Deepfake is imminent

1.8 Current Approach

It is very popular to detect Deepfake by biological characteristics in the early time because the fake videos will have unnatural blink rate and head pose. Additionally, to monitor the defects of Deepfake attacking is another branch of Deepfake detection. But with the evolution of Deepfake techniques, above schemes are no longer works. Also, some detectors are aimed on the relationships between frames. However, this method will not work for static images. All of the current approaches have a certain turnover rate. It can be predicted that the turnover rate will be increased with the development of Deepfake’s means of attack.

1.9 Dissertation’s Contributions and Organization

This dissertation provides and evaluates solutions to the discussed limitations and challenges of Section 1.4 and 1.8. Chapter 2 will discuss our approach with a new data race detector with sampling method, which introduced a new concept “Function pairs”. With the help of function pairs’ history, this approach has significantly improved the efficiency and accuracy. This approach is published and titled “AtexRace: Across Thread and Execution Sampling for In-House Race Detection”. In chapter 4, a new Deepfake defense method with embedding invisible watermaster has been proposed, which has
a 100% defense accuracy. In chapter 4, we make the conclusion and talk about our future work.
References


CHAPTER 2
THE ACROSS THREAD AND EXECUTION SAMPLING FOR IN-HOUSE RACE DETECTIONS: ATEXRACE AND ATEXRACEPLUS

This chapter presents the contributions of the race detector: *AtexRacePlus*, its preliminary version: *AtexRace*

(1) *AtexRace*: Across Thread and Execution Sampling for In-House Race Detection. Accepted, 2017, and

(2) *AtexRacePlus*: The State-of-The-Art Data Race Detector Based on Sampling Method

2.1 Summary

Data race is a major source of concurrency bugs. Dynamic data race detection tools (e.g., FastTrack) monitor the executions of a program to report data races occurring in runtime. However, such tools incur significant overhead that slows down and perturbs executions. To address the issue, the state-of-the-art dynamic data race detection tools (e.g., *LiteRace*) apply sampling techniques to selectively monitor memory accesses. Although they reduce overhead, they also miss many data races as confirmed by existing studies. Thus, practitioners face a dilemma on whether to use *FastTrack*, which detects more data races but is much slower, or *LiteRace*, which is faster but detects less data races. In this paper, we propose a new sampling approach to address the major limitations of current sampling techniques, which ignore the facts that a data race involves two threads and a program under testing is repeatedly executed. We develop a tool called *AtexRace* to
sample memory accesses across both threads and executions. By selectively monitoring the pairs of memory accesses that have not been frequently observed in current and previous executions, \textit{AtexRace} detects as many data races as \textit{FastTrack} at a cost as low as \textit{LiteRace}. We have compared \textit{AtexRace} against \textit{FastTrack} and \textit{LiteRace} on both Parsec benchmark suite and a large-scale real-world MySQL Server with 223 test cases. The experiments confirm that \textit{AtexRace} can be a replacement of \textit{FastTrack} and \textit{LiteRace}. Based on \textit{AtexRace}, we have optimized the sampling rate and the race detection and upgrade it as \textit{AtexRacePlus}, which only has the half overhead of \textit{AtexRace}.

### 2.2 Introduction

A data race (or race for short) occurs when two or more threads access the same memory location at the same time, and at least one of them is a write [16]. Race is a major source of concurrency bugs [38] and may result in real-world disasters [23][29][40].

Static race detection techniques are scalable but may report many false positives [25][37][42][51]. Various filters have been developed to address this issue. However, false positives remain and false negatives emerge with these filters in the static race detection tools [37]. Dynamic techniques report much fewer false positives. They are mainly based on either the lockset discipline [44] or the happens-before relation [16][27]. The former requires that all accesses to a shared memory location should be protected by a common set of locks. The latter [27] is usually implemented via vector clocks [16] to track the status of threads, locks and memory locations.
Happens-before based race detectors (HB detectors for short) report less false positives but incur higher overhead than the lockset-based ones. FastTrack [16], by avoiding a large number of O(n) operations on memory accesses, reduces the overhead to the level as that of the lockset-based race detectors. Even so, by continuously monitoring all memory accesses of a multithreaded program, FastTrack still incurs from 400% to 800% overhead [10][16][54].

Sampling [7][34][58] is a promising technique to reduce the overhead of dynamic detectors by selectively monitoring memory accesses. There are two types of sampling. With the assumptions that concurrency bugs cannot be eliminated during testing and daily uses of released software provide a large test bed, the first type attempts to detect races at user sites, including Pacer [7], CRSampler [12], and a possible adaption of DataCollider [14]. This type of sampling must be extremely light weight (i.e., < 5% overhead [3][26][31][59]). And they usually detect a small number of data races depending on the sampling rate and the overhead limit.

The second type aims at reducing in-house testing overhead. Before releasing a software, the developers usually test the program against a large number of test cases, and for each test case, the program may be executed multiple times. Lower overhead enables more testing and thus less races in the tested software. LiteRace [34] is a representative tool in this category. It is based on the hypothesis that undetected races often exist in cold functions that have not been frequently called. Therefore, LiteRace reduces overhead by avoiding the sampling of memory accesses in hot functions that have been frequently executed.
Figure 2.1 shows a code sketch with two threads $t_1$ and $t_2$. Functions $f_1$ and $f_2$ are repeatedly executed in $t_1$, and $f_3$ and $f_4$ are repeatedly executed in $t_2$. Races occur when $f_1$ and $f_4$ execute simultaneously, and when $f_2$ and $f_3$ execute simultaneously. Assume that $t_1$ is executed more frequently than $t_2$ and the then branches are executed more frequently than the else branches. Initially all functions are cold, but quickly $f_1$ becomes hot while other three functions are still cold. At this moment $LiteRace$ stops monitoring $f_1$ and becomes faster than $FastTrack$ because the latter still continuously monitors $f_1$. After a while $f_2$ and $f_3$ get a chance to be executed. Since both functions are cold, $LiteRace$ still monitor their executions and thus can report the race between $f_2$ and $f_3$ at a cost lower than that of $FastTrack$. Next $f_4$ is executed at the same time with $f_1$. In this case $LiteRace$ fails to detect the race between $f_1$ and $f_4$ because it already stopped tracking $f_1$. On the other hand, $FastTrack$ can catch the race because it still monitors $f_1$. This example illustrates the dilemma in choosing between full scale tools and sampling-based tools. A programmer has to either sacrifices efficiency for accuracy or sacrifices accuracy for efficiency.

We argue that programmers do not have to choose between efficiency and accuracy. This is achievable because if the following three major limitations in current sampling techniques are
addressed:

1) From the definition, a race occurrence requires two memory accesses of different threads. Therefore, sampling memory accesses in isolation is ineffective. The aforementioned example shows that a function $f$ may become hot before any other functions that race with $f$. In this case, sampling those functions that race with $f$ is useless. We call this inefficiency thread-local sampling because it does not consider any other threads when it decides whether to sample the current thread.

2) Sampling algorithms remain the same for all the executions of a program. This is ineffective because in-house testing a program is usually executed repeatedly against a large set of test cases. For a multithreaded program, a developer may even run it multiple times under a single test case. The net effect of current sampling strategy is that those functions that are cold in individual execution but hot in accumulative executions are repeatedly sampled. We call this inefficiency execution-local sampling as it does not consider previous executions when decides whether to sample the current execution.

3) The current sampling algorithms set a fixed sampling rate for entire program without considering the data race risk level of each portion of the code. A function with only local variable accesses should not be sampled because it is impossible for this function to race on the same memory location with others.

In this chapter, we proposed a new approach to address above issues and named it as $AtexRace$. $AtexRace$ is a new dynamic race detection tool based on across-thread and across-execution sampling. It is designed to sample memory access pairs from different threads and is also aware of the executions. However, several challenges must be resolved to make it practical. Firstly,
tracking memory accesses across threads incurs much larger overhead than tracking thread local data only (e.g., higher cache misses’ rate). Secondly, even if a pair of memory accesses is observed to be race-free before, it does not mean that the pair will not race later. This is because while instructions are static, the memory addresses they access are dynamic. Lastly, AtexRace avoids sampling previously observed memory pairs, which requires additional recording. With increasing number of executions, the recorded data set may grow rapidly, which may further slowdown the sampling processes (e.g., the need of more time to search memory access pairs).

AtexRacePlus is the extension of Atexrace [24]. Compare to AtexRace, it performs static analysis on each function of the program. With the information of static analysis, it customizes the sample rate for memory accesses in different functions.

The main contributions are:

- We present a novel sampling technique called AtexRace toward race detection. Unlike existing sampling techniques that are thread-local and execution-local, AtexRace is across thread and across-execution.

- To make AtexRace practical, we have designed optimization heuristics that include (1) utilizing thread-local storage to avoid competing accesses to shared sampling data set, (2) exploiting burst sampling strategy to enhance race coverage, and (3) adopting n-frequent (function) pairs to improve map lookup efficiency.

- We upgrade AtexRace to AtexRacePlus by making the following operations: 1) AtexRace treats all the memory accesses as the same. AtexRacePlus performs static analysis at the instruction level to classify all the memory accesses into 3 categories.
Each function is labeled with the counts of memory accesses in each category. 2) We propose a formula to calculate Race-Risk factor for each function pair using the knowledge gained from static analysis. The formula takes the counts of the memory accesses in each category and produce a Race-Risk factor. 3) Using the race-risk factor, \textit{AtexRacePlus} customizes the sample rate for each function. The functions with lower race-risk factor will be sampled at a lower rate. In this way, the overhead of race detection is dramatically reduced.

We have implemented \textit{AtexRace}, \textit{AtexRacePlus}, \textit{FastTrack}, and \textit{LiteRace} on top of \textit{Pintool} \cite{25}. All four tools have been evaluated on seven programs from the Parsec benchmark suite \cite{26} and a real-world large-scale program \textit{MySQL} database server. In the experiments, we run each program in the Parsec benchmark suite for 100 times and run \textit{MySQL} under 223 different test cases. The experimental results surprisingly show that \textit{AtexRacePlus} and \textit{AtexRace} detects more races in \textit{Parsec} benchmarks than \textit{FastTrack} does! \textit{LiteRace}, as predicted, detects significantly fewer races than \textit{AtexRace} and \textit{AtexRacePlus}. \textit{AtexRacePlus} and \textit{AtexRace} detects more unique races than \textit{FastTrack} and \textit{LiteRace}. In terms of efficiency, \textit{AtexRacePlus} has the lowest overhead. \textit{LiteRace} and \textit{AtexRace} reduce almost the same percentage of overhead on top of \textit{FastTrack}. This makes \textit{AtexRacePlus} a replacement of \textit{FastTrack}, \textit{LiteRace} and \textit{AtexRace}. The rest of this paper is organized as follow: section two introduces some background knowledge of concurrent programs and data race, section three describes the challenge of dynamic data race detection and our research motivation, section four presents \textit{AtexRacePlus} in details, section five describes our experiment setup and discusses the experiment result, section six compares our work with other related works,
section seven concludes the chapter.

2.3 Background

2.3.1 Multi-threaded Programs

A multi-threaded program can be defined as a tuple \( <T, \text{Lock, Mem} > \) where \( T \) is a set of threads, \( \text{Lock} \) is a set of locks (or lock/synchronization objects) and \( \text{Mem} \) is a set of memory locations (or locations for short). Each thread \( t \in T \) has a unique thread identifier, denoted as \( t.tid \).

During an execution of a multi-threaded program \( p \), each thread performs a sequence of events \( (e_1, e_2, ..., e_k) \). An event can be one of the following types. \( \text{acq}(m) \) or \( \text{rel}(m) \): synchronization events – to acquire or release a lock (Other synchronization events can be similarly defined [1]). \( \text{read}(x) \) or \( \text{write}(x) \): memory access events: to read from or write to a memory location \( x \), \( \text{call}(f) \) or \( \text{return}(f) \): control events to execute events in function \( f \) or return to execute the events from the previous function \( f \).

2.3.2 Data Race

Data races can be defined according to either the lockset discipline [10] or the happens-before relation [11]. In this chapter, we adopt the later one as it is relatively precise [1]. However, our sampling strategy is independent from concrete definitions. The happens-before relation (denoted as \( \succ \), HBR for short) is defined by the three rules [11]: (1) If two events \( \alpha \) and \( \beta \) are performed by the same thread, and \( \alpha \) appears before \( \beta \) then \( \alpha \succ \beta \) (2) If \( \alpha \) is a lock release event and \( \beta \) is a lock
acquire event on the same lock, and α appears before β then $\alpha \rightarrow \beta$, and (3) If $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$ then $\alpha \rightarrow \gamma$. Given two memory access $e_1$ and $e_2$ that access the same memory location and one of them is a write events, a race occurs if neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$.

Figure 2.2 A program with races on variable $x$ between line 8 and line 29

<table>
<thead>
<tr>
<th>Thread $t_1$</th>
<th>Thread $t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(i=1 to 2*a{</td>
<td>for(i=1 to 2*b{</td>
</tr>
<tr>
<td>if(i&lt;a) f1(i);</td>
<td>if(i&lt;b) f3(i);</td>
</tr>
<tr>
<td>else f2(i);</td>
<td>else f4(i);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>Function f1(i){</td>
<td>Function f3(i){</td>
</tr>
<tr>
<td>acq(m);</td>
<td>int p = (int)</td>
</tr>
<tr>
<td>x+=i;</td>
<td>malloc(sizeof(</td>
</tr>
<tr>
<td>rel(m);</td>
<td>int));</td>
</tr>
<tr>
<td>}</td>
<td>int q = p + 1;</td>
</tr>
<tr>
<td>Function f2(i){</td>
<td>Function f4(i){</td>
</tr>
<tr>
<td>int p,q;</td>
<td>acq(n);</td>
</tr>
<tr>
<td>p = 1;</td>
<td>x++;</td>
</tr>
<tr>
<td>q = p +1;</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.2 shows a multi-threaded program $p$ that extends the code sketch given in Figure 2.1. The program consists of two threads $t_1$ and $t_2$ operating on shared variables $x$. There are two locks $m$ and $n$ protecting accesses to shared variables $x$. Given two parameters $<a, b>$ thread $t_1$ consecutively calls function $f_1$ for $a$ times and then calls function $f_2$ for $a$ times within a loop (lines 1–4); and thread $t_2$ performs similar calls to functions $f_3$ and $f_4$ each for $b$ times (lines 17–20). The four functions $f_1$ and $f_4$ increase the values of $x$ based on the parameters passed to them. Function $f_2$ has some local variable operations, while Function $f_3$ has a malloc operation besides local variable operations. Due to the parallel execution of the two threads in Figure 2.2, any pair of functions between threads $t_1$ and $t_2$ can potentially be executed simultaneously. The four pairs of
functions that can be executed at the same time are \( <f_1, f_3>, <f_1, f_4>, <f_2, f_3>, \) and \( <f_2, f_4> \). For the pairs \( <f_i, f_i> \), as the variable \( x \) is protected by different locks (i.e., lock \( m \) in function \( f_1 \) but lock \( n \) in function \( f_i \)), races may occur. For example, if line 8 and line 29, are executed at the same time, the program may produce incorrect results due to the race on variable \( x \).

2.4 Motivation

2.4.1 Dynamic Data Race Detection

Dynamic data race detectors use instrumentation techniques to detect data races at run-time. Dynamic instrumentation tools [25] allow the user to insert code to original program to perform analysis. Figure 2.3 explains how to instrument a program to detect data race. \( x + = 1 \) is compiled to a low-level representation as three memory accesses. \( \text{onRead} \) and \( \text{onWrite} \) are so called analysis routines. Analysis routine calls: \( \text{onRead}(x) \) and \( \text{onWrite}(x) \) are injected right after the read and write operations to \( x \) respectively. \( \text{onRead}(x) \) and \( \text{onWrite}(x) \) record the memory accesses to \( x \) using shadow memory technique [27] and perform HBR violation check [11]. If there is an HBR violation, a data race will be reported. Dynamic race detectors usually incur large overhead [1], [22] due to the operations we mentioned previous. The example in Figure 2.3 also illustrates the overhead. For each access to the location \( x \), one analysis routine call such as \( \text{onRead}(x) \) or \( \text{onWrite}(x) \) is inserted [1]. Analysis routine calls brings in two types of operations that cost time [1], [16], [28].

The first type is shadow memory operations (or meta data [1], [16]). For each memory location,
all accesses to it are tracked by analysis routine calls. Access information at additional memory
locations are stored as shadow memory (e.g., \texttt{shadowMemory(x)} in Figure 2.3). Similarly, extra
information is stored for each thread, known as shadow threads (e.g., \texttt{shadowThread(t)} in Figure
2.3). Therefore, shadow memory and shadow threads cause extra memory allocation and extra
memory accesses (e.g., \texttt{Sx} and \texttt{St} for memory location \texttt{x} and thread \texttt{t}, respectively, in Figure 2.3).
Those operations are inevitable and incur large overhead. Many instrumentation frameworks
already provide fast shadow threads interfaces (e.g., Thread Local Storage in \textit{Pin} [25] and Thread
Execution Blocks supported by Windows OS [29]). To the best of our knowledge, there is no fast
way to access shadow memory. For Java program, the speed of accessing to the shadow memory
could be faster if it is allocated together with the associated memory allocation in the original
program [28]. However, for C/C++ programs, this becomes difficult.

![Dynamic Data Race Detection]

\texttt{onRead(x)} // or \texttt{onWrite(x)}
\begin{align*}
x & += i; \\
tmp & = x; \text{onRead}(x); \\
tmp & += i; \\
x & = tmp; \text{onWrite}(x);
\end{align*}

Figure 2.3 An illustration on the instrumentation and race detection for each memory access.
The second type of operations in analysis routine is HBR violation checking. After fetching data from shadow memory and shadow threads, analysis routines check to see whether any HBR violation occurs and report them as data races. This type of operations also incurs overhead because of additional memory accesses and HBR violation pattern matching, especially the write operations to maintain the access information (i.e., to update $S_x$ in Figure 2.3). Note that, FastTrack optimizes the process on race detection, but it still requires maintenance (read and write) on shadow data.

2.4.2 Sampling on Dynamic Data Race Detection

The heavy overhead of dynamic data race detection is unavoidable because it mainly comes from shadow memory operations and HBR violation pattern matching operations. Therefore, several sampling approaches have been proposed to reduce the run-time overhead by tracking a subset of the events and detect races among them. Comparing to detecting data race on all the events, overhead is reduced because there will be less shadow memory operations and HBR violations pattern matching operations. Existing sampling approaches can be classified into deployed sampling [15], [22] and in-house sampling [18], [23]. The former type of approaches is deployed at the users’ sites after a software is released. Such approaches are based on the crowd-source testing: if there are many users, races escaped during inhouse testing may be detected by sampling a tiny portion of an execution by each user. Hence, deployed sampling requires extremely low run time overhead that is usually limited to 5% [15]. The latter attempts to reduce runtime
overhead during in-house testing phase. The representative tool is LiteRace [16]. Our work falls into this category. In the rest of this subsection, we use LiteRace as an example to describe the in-house sampling technique in detail.

During the execution of the program, many code regions are being executed more than one time. For example, a function could be called many times during the execution. LiteRace is based on the cold-region hypothesis: races are likely to occur when a thread is executing a cold region (i.e., the program portion not frequently executed). Based on this hypothesis, LiteRace tries to avoid analysis routine calls for frequently executed functions (i.e., hot functions). Initially, thread-local sampling rate of each function is set to 100%. Shadow memory operations and HBR violation pattern match are performed for each memory access every time the function is called. This

Figure 2.4 Four executions scenarios of the program in Figure 2
sampling rate is then gradually reduced whenever a function is called by the corresponding thread. If the sample rate is not 100%, race detection operations will be skipped sometimes for the function.

The sample rate keeps decreasing until a low bound is reached (e.g., 0.1%). For example, in Figure 2.2, LiteRace initially checks all events from function $f_1$. After the function is executed once, the thread-local sampling rate of function $f_1$ by thread $t_1$ is reduced. If thread $t_2$ calls function $f_3$, the sampling rate of function $f_3$ by thread $t_2$ is also reduced in the same way.

2.4.3 Limitations of Existing Sampling Approaches

LiteRace reduces overhead with the sacrifice of race detection capability. For example, in an evaluation, it only detected about 70% of frequent data races and about 50% of rare data races of the tools, such as FastTrack, which monitors every event during the executions. The result is also verified by other works [15]. The result is caused by the limitation of LiteRace. We explain the limitation of LiteRace using our motivation example in Figure 2.2. Figure 2.4 gives four execution cases that illustrate how the functions in the two threads interleave. In each case, a column shows the execution of a thread in term of function calls. The difference between the four cases is at how the last call to function $f_1$ and the first call to function $f_2$ by thread $t_1$ interleaves with the last call to function $f_3$ and the first call to function $f_4$ by thread $t_2$. Recall that locks $m$ and $n$ protect the accesses to $x$ in functions $f_1$ and $f_4$, respectively. Because two different locks are used, a race on variable $x$ occurs when functions $f_1$ and $f_4$ execute in parallel. No data race occurs in case (a), case (b) and case (d). For case (a) and case (b) because neither pair of functions may execute in parallel. That is, we can infer that accesses in function $f_1$ happen before accesses in function $f_4$ by
following lock acquisition order (i.e., the solid arrows) and the program order within each thread (i.e., the dashed arrows). As for case (d), even $f_2$ and $f_3$ may execute in parallel, $f_2$ only has local variable operations, so there is impossible to occur race. However, for case (c), there is no strict order between the accesses in functions $f_1$ and $f_4$; hence, a happens-before race detector may detect the race on $x$ from the two functions.

When LiteRace is applied to the four cases in Figure 2.4, a function is not tracked after it has been called by the same thread for certain number of times. Therefore, function $f_1$ executed by thread $t_1$ and $f_3$ executed by thread $t_2$ are no longer tracked once they become hot functions. In case (c), even when function $f_1$ and function $f_4$ execute in parallel, LiteRace may miss the race. This is because LiteRace only tracks the cold function $f_4$ without tracking function $f_1$.

We believe the main reason that LiteRace frequently fails to detect races, as observed previously [15], is that its sampling across threads is not coordinated. Since a data race requires two conflict memory accesses from two threads, sampling one memory access from one thread but not the other is useless. This is illustrated by cases (c) above. Consider an extreme case where all races involve a function. If this particular function is considered hot after being visited several times, all future samplings are in vein.

Besides the issue of thread-local sampling, LiteRace also suffers from execution-local sampling. When testing a multithreaded program by running it repeatedly against a large number of test cases, the same thread interleaving, with minor variations, tend to be exercised since thread schedulers generally switch among threads at the same program locations. In addition, although the whole program execution may witness variants from one run to another, partial execution may
exhibit similar behavior. For example, even all the four cases in Figure 2.4 are executed in different runs, the initial interleaving of two threads are similar. That is, functions \( f_1 \) and \( f_3 \) interleave until functions \( f_2 \) or \( f_4 \) is called. We highlight these function calls in grey background for illustration purpose. As LiteRace is unaware of execution similarities, it adopts the same sampling strategy across different executions. The net effect of strategy is that those functions that are cold in individual execution but hot in accumulative executions are repeatedly sampled. This defeats the principle of sampling that the real cold cases should be tracked.

Lastly, the current sampling techniques set a fixed sample rate across the board for the entire program without evaluating the race-happen possibility in each part of the program. For example, in Figure 2.2, function \( f_2 \) only has memory accesses to the local variable \( p \) and \( q \). Since those 2 variables are not thread-shared variable. There is no need to sample this function. We believe setting flexible sampling rate could significantly improve race detection efficiency. Besides, setting flexible sampling rate makes it possible to have a trade-off ability between accuracy and speed by setting sampling rate bound. However, current sampling techniques do not provide such kind of method. The three main limitations of current sampling techniques motivate our work in this chapter.

2.5 Our Approach

2.5.1 Goal and Challenge

In this section, we present our approach to address the three limitations of current sampling techniques.
1) Thread local sampling
2) Execution local sampling
3) Same sample rate across the entire program

To address the limitation of thread-local sampling, our insight is that whether to sample a memory access event should also depend on the execution of other threads and those already observed executions. That is, even if a memory address has been accessed by a thread many times, we may still need to sample it if a second thread access the memory address for the first time. As for execution-local sampling, our idea is to keep and store sampling information from previous runs. Except the first execution that starts with a cold run, the subsequent executions load sampling information of accumulated prior executions. Although such approach incurs overhead, we believe less sampling with optimization heuristics can lead to net benefit. The new sampling approach, \textit{AtexRace}, also works at function levels like \textit{LiteRace}. But unlike \textit{LiteRace}, \textit{AtexRace} mainly samples accesses inside a pair of functions whose simultaneous executions are not observed before, including previous executions. As for the limitation of the same sample rate for all the functions, our solution is to evaluate the possibility of data occurrence for each function by performing static analysis on each function and assign different sample rate for each function based on data race risk level. Function with high risk will be sampled at a higher rate. Lower sample rate for the functions that data races are less likely to exist. Functions without memory accesses to the thread-shared variables will not be sample at all. We also applied this method to \textit{Atexrace} which make it evolves to \textit{AtexRacePlus}.

In the rest of this section, we first talk about \textit{AtexRace}, then its upgraded version of
AtexRacePlus and we will discuss how to address the limitation of existing sampling algorithm.

2.5.2 Basic AtexRace Algorithm

The overview of AtexRace is shown in Figure 5. During execution, when function f_y in thread t_y is being executed, AtexRace collects all the functions (e.g., f_x and f_z) that are being executed by other threads. By doing so, AtexRace forms pairs of functions that are being executed simultaneously (e.g., (f_x, f_y)). It then makes a sampling decision according to whether a pair of functions have been executed in parallel before. If so, neither function is sample; otherwise, both are sampled. If a function is sampled, all its events are passed to a race detector. At the end of an execution, all function pairs are saved and will be used in the next execution. Note, in order not to report false positives, all synchronizations are fully sampled. This is the same as LiteRace.

Algorithm 1 gives the basic AtexRace algorithm that takes a program p and a set of function pairs FPair that have been observed in the previous executions. The first three lines initialize two necessary runtime data structures: a map F that maintains the functions being executed by each thread, and a map S that indicates whether memory accesses from a thread should be sampled. Both F and S are empty initially.

The function onCallFunc (lines 5–19) is the core of our sampling. Whenever a function f is to be executed (i.e., at the entrance of function f) by a thread t, for every other thread t' in program p, AtexRace checks whether the pair (f, F(t')) already exists in FPairs. If not, S is updated to map both threads t and t' to true; otherwise, S maps t to false. A true value of S(t) mandates sampling
of the current memory access in thread \( t \) and a false value does the opposite. Next, \textit{AtexRace} executes events in function \( f \) (line 14) and samples its memory accesses (i.e., function \textit{onMemoryAccesses}) if \( S(t) \) is true. At the end of the call to function \( f \), \textit{AtexRace} merges \( FPairs \) and the observed function pairs \( \langle f, F(t') \rangle \), which indicates that the function \( f \) and another function \( F(t') \) in thread \( t' \) have been executed simultaneously.

In practice, two functions from different threads are usually called at different time. Therefore, it is the case that, a function \( f \) is initially not sampled but later it should be sampled as a different thread \( t' \) calls a function \( f' = F(t') \) and the pair \( \langle f, F(t') \rangle \) is never observed before. This is considered by \textit{AtexRace}. We can see from lines 10 and 11 that at the call entrance to function \( f' \), thread \( t' \) also performs an iteration over other threads at line 7. At the iteration on thread \( t \), it cannot find the pair in \( FPairs \). Then it maps both threads \( t' \) and \( t \) to be true value in structure \( S \). So, the function \( f \) executed by thread \( t \) has to be sampled.
Algorithm 1: Basic AtexRace

**Input:** \( p \) – a multithreaded program.

**Input:** \( FPairs \) – a set containing functions.

1. Let \( F \) be an empty map from a thread to a function
2. Let \( S \) be a map from a thread to a Boolean value.
3. For each thread \( t \subseteq p \), \( F(t) := \emptyset \), \( S(t) := \text{true} \) end for
4. 
5. Function \( \text{onCallFunc}(\text{Thread } t, \text{ Func } f) \)
6. Let \( F(t) := f \) and \( S(t) := \text{false} \) //\( S(t) \) is a temporary variable that keeps \( S(t) \)
7. For each thread \( t' \in p \), \( t \neq t' \) do
8. Pair := \( \langle f, F(t') \rangle \)
9. If pair \( \notin FPairs \) then
10. \( S(t) := \text{true} \)
11. \( S(t') := \text{true} \)
12. End if
13. End for
14. \( S(t) := S(t) \)
15. Execute \( f \)
16. For each thread \( t' \in p \), \( t \neq t' \) do
17. \( FPairs := FPairs \cup \{ \langle f, F(t') \rangle \} \)
18. End for
19. End Function
20. Function \( \text{onMemoryAccess}(\text{Thread } t, \text{ Event } e) \)
21. If \( S(t) = \text{true} \) then
22. Call data race detector
23. End if
24. End Function
25. Save \( FPairs \)

2.5.3 Limitations of Basic AtexRace

The basic sampling algorithm of AtexRace suffers from the two limitations: (1) given two
function $f_1$ and $f_2$, even if their parallel execution has been observed and tracked (thus become hot), races between them may still not detected; and (2) significant overhead resulted from across thread and execution sampling.

The first limitation is the issue of Race Coverage. A function usually contains multiple basic blocks (BBLs). An execution of a function does not mean all its BBLs are executed. For example, Figure 2.5 shows two functions $f_5$ and $f_6$ that contain two races on variables $x$ (lines 6 and 21) and $y$ (lines 18 and 9). There are four BBLs $b_{11}$, $b_{12}$, $b_{21}$, and $b_{22}$ (we omit other BBLs in the if statement for simplicity). Since the two threads in the example execute $f_5(10)$ and $f_6(100)$, respectively, only $b_{11}$ and $b_{22}$ are executed. Hence, the race on variable $x$ (lines 6 and 24) is detected while the race on variable $y$ (lines 19 and 10) is not. If the pair $\langle f_5, f_6 \rangle$ is considered hot after this execution, the race on $y$ can never be detected by the basic $AtexRace$. One approach to address this issue is to degrade the sampling level from functions to BBLs and then apply either $LiteRace$ or the Part 1 of our $AtexRace$. However, this bring heavy runtime overhead and may even incur more overhead than a full detector such as $FastTrack$. This is because, compared to a function, a BBL usually contains much fewer instructions. As a result, the sampling overhead (in time) per BBL may already larger than the race detection overhead without sampling. Because sampling algorithm is not extremely lightweight, it is not worthy to perform sampling at BBL level.
On the other hand, for C/C++ programs, even an instruction contains one or more memory accesses, it is possible that each execution of the instruction may accesses different memory location. For example, considering the following two lines of code:

1. `Object obj = &getObj(...);`
2. `obj->val ++;`

We can observe that, within the same and repeated executions of the two lines, if the pointer obj points to different objects, it accesses different memory locations at line 2. Therefore, for sampled memory accesses, it is still necessary to track them.

The second limitation is the Sampling Overhead of AtexRace itself. A sampling tool should sample as fewer memory accesses as possible to reduce the overhead. At the same time, it should also try to incur less overhead from its sampling strategy. LiteRace adopts thread-local sampling and requires two thread-local counters per-function. This can be efficiently implemented [34].

For AtexRace, there are expansive map queries (i.e., FPairs) on each function call (lines 9–10). These operations bring heavy slowdown for two reasons. Firstly, with the increasing number
of function calls by multiple threads, the size of $FPairs$ also increases, resulting in a large data set. For example, in our experiment, after 223 executions on MySQL, there are nearly 70,000 function pairs. A query over such a large map is time consuming. Secondly, the map $FPairs$ is accessed by multiple threads. This requires synchronizations among different threads when they operate on the map. Such synchronization incurs further slowdown. Besides, when different threads access the map $FPairs$, the cache miss rate will be higher because once a thread updates the map, all other threads that query the map must wait until their local caches are updated. This again leads to additional time consumption. All these reasons bring challenges to reduce the overhead of our sampling algorithm $AtexRace$ itself.

2.5.4 Optimizations

Algorithm 2 is an enhancement to the basic $AtexRace$ algorithm that addresses the two kinds of limitations.

To address the issue of race coverage, $AtexRace$ further samples those sampled function pairs in order to increase their coverage on data race detection. This corresponds to lines 18–24 in Algorithm 2. For this part, $AtexRace$ accepts a sampling rate (i.e., the input $r$ to Algorithm 2) and samples the function pair according the rate. Note that, $AtexRace$ does not perform a simple sampling that generates a random number and compares the random number with the given sampling rate. Instead, $AtexRace$ adopts burst sampling strategy [34]. It samples the first $n$ consecutive calls out of all $m$ calls to a function such that the rate $(n ÷ m) × 100\%$ equals to the
given sampling rate \( r \). For example, if the sampling rate is 10\%, it samples the first 10 calls and discards the next 90 calls to the same function, resulting the sampling rate of 10\%. Of course, to implement this functionality, a counter mapped from each function pair is required. Hence, the original set of function pairs is changed into a map (see the fourth input and the lines 18, 19 and 29 in Algorithm 2).

To overcome the second kind of limitations, we firstly propose to use thread-local maps. In Algorithm 2, we use the symbol \( FP \) to denote the thread-local maps of function pairs. That is, we allocate one map structure for each thread; and when \( AtexRace \) starts an execution, it duplicates the given map data (line 7). During an execution, \( AtexRace \) only checks whether the pair exists in the map \( FP \) of the current thread (lines 14 and 19). If a pair already exists in a thread-local map, its counter is incremented by 1 at line 18. At the end of an execution, \( AtexRace \) merges all thread-local maps and saves the merged map (lines 39–43). Secondly, we do not record all function pairs observed in previously executions. Instead, we only keep the recently frequently observed function pairs. Given an execution \( e \) and a number \( n (n \geq 1) \), we define a function pair \( \langle f_x, f_y \rangle \) to be \( n \)-frequent with respect to execution \( e \) if \( \langle f_x, f_y \rangle \) is observed in current and all the \( n-1 \) previous executions. Specially, when the value of \( n \) is 1, the 1. frequent function pairs are those observed in the current execution. By keeping only, the \( n \)-frequent function pairs, the recorded function pairs are those frequently executed. This is reasonable not to sample these frequent function pairs to reduce sampling overhead. Hence, for each execution, the number of function pairs taken as input is small and does not increase with increasing number of executions. The third and the fourth inputs to Algorithm 2 reflects this design, where \( n \) determines the function pairs in \( FPairs \). By adopting
thread-local maps and recording only n-frequent function pairs, the only side effect is that 
_AtexRace_ may sample function pairs that have been sampled in the same execution due to the 
content difference of different threads within the same execution. This may incur unnecessary 
overhead. However, it produces no bad result on the data race coverage as sampling the same 
functions more than one time also increases the probability to detect those missed data races (see 
the first kind of limitations in last section).
Algorithm 2: Complete ArxRace

Input: p – a multithreaded program.
Input: r – a sampling rate.
Input: n – a number determine n-frequent value
Input: FPairs – a map (from functions pairs to counters) of the last r - 1 executions.

/* Initialization */
1. let F be an empty map from a thread to a function
2. let S be a map from a thread to a Boolean value.
3. let FP be a map from a thread to a copy of FPairs. //thread-local maps
4. for each thread t ∈ p do
   5. F(t) := Ø
   6. S(t) := true
   7. FP(t) := FPairs
end for

/* Runtime Sampling */
10. Function onEnterFunc(Thread t, Func f)
11. let F(t) := f and St := false //St is a temp variable that keeps S(t)
12. for each thread t’ ∈ p, t ≠ t’ do
   13. pair := (f, F(t’))
   14. if pair ∉ FP(t) then
      15. St := true
      16. S(t’) := true
   else
      17. FP(t) := FP(t) ∪ {pair, Counter(FP, pair) + 1}
      18. if counter(pair, FP(t)) satisfies r then
         19. St := true
         20. S(t’) := true
      else
         21. St := false
      end if
   end if
26. S(t) := St
27. execute f
29. for each thread t’ ∈ p, t ≠ t’ do
30. FP(t) := FP(t) ∪ {pair, 1}
end for

end Function
13. Function onMemoryAccess(Thread t, Event e)
34. if S(t) = true then
   35. call data race detector
end if
end Function

/* The End of an Execution */
38. Let FPairs’ be an empty map.
40. for each thread t ∈ p do
41. FPairs’ := FPairs’ ∪ F(t)
end for
42. save FPairs’
### 2.5.5 *AtexRacePlus* Algorithm

The overview of *AtexRacePlus* is shown in Figure 2.6. Given a program, during the run time, *AtexRacePlus* will monitor all threads. If a function f1 starts to execute in thread t1, *AtexRacePlus* collects all the functions (e.g., f2 and f3) that are being executed in other threads. *AtexRacePlus* forms pairs of functions being executed simultaneously in different threads (e.g., <f1, f2>, <f1, f3>). The sample rate of each function pair is calculated with the knowledge gained from static analysis. Then, it makes a sampling decision according to how frequently the function pairs have been executed in parallel before. If the pair has been frequently observed before, neither of functions will be sampled; otherwise, both are sampled. The sample rate defines the threshold of at which point the function pair is considered frequently observed. If a function mark as” sample”, all its events are passed to a race detector. At the end of an execution, all function pairs are recorded and will be used in the next execution if any.

Note that, in order not to report false positives, all threads synchronization operations are fully sampled. Based on our design, the scheme has been split into two parts to represent:

- Sample rate calculation algorithm, which produces a flexible sampling rate.
- Main algorithm aims to introduce the core part of *AtexRacePlus*.

### Sample Rate Calculation Algorithm

The existing sample techniques sample all the code regions at the same sample rate. High
sample rate will lead to high overhead on data race detection while low sample rate will sacrifice data race detection capability. Based on our observation, we find previous work: \textit{AtexRace}, which still has a huge potential to improve. First, data race only happens between functions that contain memory accesses to the thread-shared memory objects. If a function has no memory access to shared memory objects, it is not even necessary to be sampled. In other word, the sample rate for this function should be set to 0, and the sample rate for those two functions of the pair will be set to 0, too.

We classify all the memory accesses into three categories: global variable access, heap variable access and stack variable access. Global variables are defined out of any functions, has the global scope and its lifetime is as long as the lifetime of the program execution life cycle. The heap variable is defined using the memory allocations function such \textit{malloc()}, \textit{new()}, \textit{free()} and \textit{delete()}. Stack variables, namely the local variables, which are stored in stack regions and cannot be shared by the other threads. The lifetime of a stack variable ends after the function finishes. Global variables are absolutely thread shareable while stack variables are absolutely not thread shareable. Heap variable can be accessed by two scenarios, one is it could be accessed by other threads via pointers, the other scenario is if a new malloc called by thread \( t_1 \) without any available arena right now, then this malloc will be assign to the last used arena, suppose this is using by \( t_2 \), now \( t_1 \) and \( t_2 \) will share the memory location and a data race could happen at this time. By statically analyzing the memory accesses in each function, it is possible to predict the possibility of data races occurrence. With the predicted possibility, the sample rate is customized for each function. If the function only has stack variable accesses, the sample rate will be set to 0. The lower sample
rate for the functions happened only if they have heap variable accesses but no global variable access. Otherwise, the sample rate won’t be modified.

Algorithm 3 explains how to assign sample rate to each function. The sample rate of each function equals $r \times f_{\text{actor}}$ where $r$ is an upper bound of the sample rate. $f_{\text{actor}}$ is calculated based on the number of each memory accesses type in the function $f$. If $f$ contains at least one global variable, the $f_{\text{actor}}$ is set to 1, that is to say, the sample rate of this function equals to the sample rate upper bound $r$. If this function neither contains global variable access nor heap variable access, the factor is set to 0 due to it only has stack variable memory accesses. No race can possibly happen in a function only contains local variable memory access, so the sample rate is set to 0 to avoid unnecessary overhead. All the memory accesses in this function will not be sampled. In the last
case, the function doesn’t contain the global variable access but do contains heap variable accesses. In some scenario, the heap variable operations can cause race as we mentioned above. And the function has more heap operations, it will have more chance to share the same arena with other thread. So, we set factor to the ratio along with the number of heap variable accesses in the function over total number of memory accesses in the function. In other words, if 50% of memory accesses are heap variable accesses, factor will be set to 0.5 and the sample rate equals 0.5 * r. The sample rate for the function pair <f₁, f₂> is defined as the min(rate(f₁), rate(f₂)).
Main Algorithm

Algorithm 4 gives the AtexRacePlus algorithm that takes three inputs: a multi-threaded program \( p \) and a set of function pairs \( FPair \) which was created by previous executions, and a
sample rate upper bound $r$. The first three lines initialize three necessary run-time data structures: a map $F$ that maintains the functions currently being executed by each thread, and a map $S$ that indicates whether memory accesses from the current thread should be sampled, and the $FP$ is the copy of $FPairs$ as the thread local map. Forth line’s Rate is a map from each function to its sample rate. Both $F$ and $S$ are empty initially. Rate is constructed using Algorithm 3. The function $onEnterFunc$ (lines 11–33) is the core of our sampling. Whenever a function $f$ is to be executed (i.e., at the entrance of function $f$) by a thread $t$, $AtexRacePlus$ will monitor every other thread $t'$ in program $p$ to check whether the pair $<f, F(t')>$ already exists in $FP$, and check whether function $f$ only has local variable access. If both are not, $S$ is updated to map both threads $t$ and $t'$ to true; otherwise, goes to else branch. A true value of $S(t)$ mandate sampling of the current memory access in thread $t$ and a false value does the opposite.

In else branch, the upper bound $r$ will be customized. Rate is a map from each function to its sample rate. In line 20 to 25, the value of $S(t)$ is determined by the condition whether counter ($<pair, FP(t)>$) satisfies $r$. It means the sample rate is set to $r$ where $r$ is the sample rate of the function pair $<f, F(t')>$ and equals to the smaller sample rate of the two functions. If $r$ is satisfied, $S(t)$ and $S(t')$ will be set to true, else $S(t)$ will be set to false. The. The mechanism that value of $S(t)$ is determined by the function pair $<f, F(t)>$ addresses the limitation of thread-local sampling. Even function $f$ is already observed many times. If function pair $<f, F(t)>$ has never been observed, we should still sample $f$. $AtexRacePlus$ will be unlike to miss races in case $c$ of Figure 2.4 because even all $f_1, f_2, f_3$ and $f_4$ are considered as hot region after several iterations, they will still be sampled if $<f_1, f_4>$ is observed at the first time.
Next, *AtexRacePlus* executes all instructions in function $f$ (line 29) and samples its memory accesses (i.e., function `onMemoryAccesses`) if $S(t)$ is true. At the end of the call to function $f$, *AtexRacePlus* merges $FPairs$ and the observed function pairs $<f, F(t')>$, which indicates that the function $f$ and another function $F(t')$ in thread $t'$ have been executed simultaneously.(line 30-32)

Note that, in practice, two functions from different threads are usually called at different time. Therefore, it is usually the case that, a function $f$ is initially not sampled but later it should be sampled as a different thread $t'$ calls a function $f' = F(t')$ and the pair $<f, F(t')>$ is never observed before. This is also considered by *AtexRacePlus*. We can see from lines 16 and 17 that at the call entrance to function $f'$, thread $t'$ also performs an iteration over other threads at line 13. At the iteration on thread $t$, it cannot find the pair in $FPairs$. Then it maps both threads $t'$ and $t$ to be true value in structure $S$. Then, the function $f$ executed by thread $t$ has to be sampled.

The last step in Algorithm 4 (lines 39-41) is to keep and store all the observed function pairs. Those data will be passed to *AtexRacePlus* as parameter in the next run. This mechanism makes *AtexRacePlus* a cross-execution sampling technique. It keeps the sample information cross different runs. *LiteRace* does not record the sample information so that for every execution, all the functions are considered as cold regions at the beginning. During this execution, a function is not sampled if it is frequently observed. The loss of sample information may cause more overhead because some functions may already be observed many times in the previous executions.
Algorithm 4: AteXRacePlus

Input: p—a multi-threaded program
Input: r— is a sampling rate upper bound
Input: FPairs—a map (from functions pairs to

counters) of all the previous executions.

// Initialization
1 let F be an empty map from a thread to a function
2 let S be a map from a thread to a Boolean value
3 let FP be a map from a thread to a copy of
   FPairs.//thread-local maps
4 let Rate be a map from a function to a sample rate.
5 Rate ← CalculateSampleRate(p, r)
6 for each thread t ∈ p do
7     F(t) = ∅
8     S(t) = true
9     FP(t) = FPairs  //deep clone
10 end

// Calculate sample rate for each function
11 Function onEnterFunc(Thread t, fun f):
12     let F(t) = f and S(t) = false
13     // S(t) is a temporary variable keeps S(t)
14     for each thread t ′ ∈ p, t ′ ≠ t do
15         pair = ⟨f,F(t ′)⟩
16         if pair ∈ FP and Rate(f) ≠ 0 then
17             S(t) ← true
18             S(t ′) ← true
19         else
20             FP(t) = FP(t) ∪ {⟨pair, Counter(FP, pair) + 1⟩}
21             r = Min(Rate(f), Rate(F(t ′)))
22             if counter((pair,FP(t ′))) satisfies r then
23                 S(t) = true
24                 S(t ′) = true
25             else
26                 S(t) = false
27             end
28     end
29     S(t) ← S(t)
30     execute f
31     for each thread t ′ ∈ p, t ′ ≠ t do
32         FP ← FP ∪ {⟨f,F(t ′)⟩}
33     end
34 End Function

Function onMemoryAccess((Thread t, Event e)):
35     if S(t) == true then
36         call data race detector
37     end
38 End Function

// Save function pairs
39 for each thread t ∈ p do
40     FPairs = FPairs ∪ FP
41 end
42 Save FPairs
2.5.6 AtexRacePlus on Example Program

In this section, we use the running example in Figure 2.2 to illustrate how AtexRacePlus sampling its executions. Initially, AtexRacePlus samples both functions \( f_1 \) and \( f_3 \) as the input \( FPairs \) are empty. Such sampling continues until in each thread the recorded functions pairs contain \(<f1, f3>\). Probably after a certain number of calls to both functions, AtexRacePlus stops continuous sampling of \( f_1 \) and \( f_3 \) because \(<f1, f3>\) is hot. Of course, in our algorithm, functions in a hot pair still have chances to be sampled due to our burst sampling strategy. Next, suppose thread \( t_1 \) calls \( f_2 \) for the first time while \( t_2 \) is executing \( f_3 \). Because pair \(<f2, f3>\) is cold, AtexRacePlus restarts to sample function \( f_2 \). Of course, \( f_3 \) is sampled as well. Similarly, AtexRacePlus restarts to sample function \( f_1 \) if functions \( f_1 \) and \( f_4 \) are executed at the same time. On the other hand, if it is \( f_2 \) and \( f_4 \) that are executed at the same time, neither \( f_1 \) nor \( f_3 \) is sampled.

Therefore, in Fig 2.4, for cases (c), AtexRacePlus has larger probability to detect the races that are probably missed by LiteRace. However, for cases (a) and (b), although no race can be detected, AtexRacePlus still samples the first calls to function \( f_3 \) and \( f_4 \). In the subsequent execution, after functions \( f_3 \) and \( f_4 \) are called for several times, AtexRacePlus stops the continuous sampling of the two functions. After one execution of the example program, AtexRacePlus records the observed function pairs (probably the four pairs: \(<f1, f3>\), \(<f1, f4>\), \(<f2, f3>\), and \(<f2, f4>\). If the program is executed again, AtexRacePlus may not continuously sample the function pairs already collected.
Table 2.1 Sample rate of each function pairs in the program of Figure 2

<table>
<thead>
<tr>
<th>Function</th>
<th>$f_1$</th>
<th>$f_2$</th>
<th>$f_3$</th>
<th>$f_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Access#</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Heap Access#</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Stack Access#</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Factor</th>
<th>1</th>
<th>0</th>
<th>0.8</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Pair</td>
<td>$\langle f_1, f_3 \rangle$</td>
<td>$\langle f_2, f_4 \rangle$</td>
<td>$\langle f_1, f_4 \rangle$</td>
<td>$\langle f_2, f_3 \rangle$</td>
</tr>
<tr>
<td>Factor</td>
<td>0.8</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

| Sample Rate       | $0.8 \times r$ | 0     | $r$   | 0     |

Hence, the total overhead to detect data race can be reduced, not only within the same execution but also across different executions of the same program.

All the function pairs will be sampled at different sample rate. The sample rate is determined by the knowledge obtained from static analysis using the formula given in Algorithm 3. Table 2.1 shows the sample rate factor of each function pairs in the multi-threaded program example in Figure 2.2. The factor of $\langle f_2, f_3 \rangle$ and $\langle f_2, f_4 \rangle$ is 0 because there are only stack accesses in $f_2$. It is impossible for $f_2$ to race on the same memory location with other functions. There is no need to sample those pairs. For function pair $\langle f_1, f_3 \rangle$ the sample rate is set to the smaller one of the two functions. As for the $\langle f_1, f_4 \rangle$, because of global variable existing in both, the sample rate won’t be lowered.

2.5.7 Discussion on AtexRacePlus

We aim to reduce race detection overhead without sacrificing race detection capability when there are many test cases. AtexRace does not target a single execution as one of our innovations is
to record the recently observed function pairs and skips their sampling in subsequent executions. Hence, on a small number of executions, it may initially incur larger overhead than that by FastTrack and LiteRace. AtexRace is more suitable for programs (e.g., industrial programs) that are tested against a large number of test cases. Of course, as a dynamic sampling approach, it also reports false negatives.

Figure 2.7 Ideal overhead changes with increasing executions.

Figure 2.7 shows the ideal scenario of AtexRace. Initially, AtexRace may incur higher overhead than LiteRace or even FastTrack. However, with increasing number of executions, AtexRace gradually incurs lower overhead.

FastTrack is an effective data race detector; the sampling methods could be implemented on top of it. As we discussed before, there are lot of limitations exists in current sampling method. And our previous approach, AtexRace has solved the major limitations, but it still has a potential optimization space.
\textit{AtexRace} use the same sampling rate for all the sampled function regardless the type of memory accesses, which is not efficiency. The higher sampling rate could perform more detection accuracy, meanwhile, the large overhead also is been brought. It is more suitable for the functions have a higher race probability. Oppositely, the lower sample rate has a lower overhead, but may miss a lot of data races. We believe if the lower race probability functions could have a lower sampling rate, while the higher race probability could have a higher sampling rate, the detection tool could have a better speed performance without sacrificing detection accuracy. And this method has been used to \textit{AtexRacePlus}. Even more, \textit{AtexRacePlus} will not detect the functions only have local variables. This is the main difference between \textit{AtexRace} and \textit{AtexRacePlus}.

The same idea, during the race detection process, \textit{AtexRace} will monitor all the instructions in the function. But \textit{AtexRacePlus} will skip the one only has local variable accesses. This optimization could reduce a significant overhead. And it should be applicable to FastTrack and LiteRace, too.

For \textit{AtexRace}, there are expansive map queries (i.e., \textit{FPairs}) on each function call (e.g., lines 15 in algorithm 2). These operations bring heavy slowdown for two reasons. Firstly, with the increasing number of function calls by multiple threads, the size of \textit{FPairs} also increases, resulting in a large data set. For example, in our experiment, after 223 executions on MySQL, there are nearly 80,000 function pairs. A query over such a large map is time consuming which may be more than the time saved by sampling method. This runs in the opposite direction of our original intention. In order to reduce the overhead of data retrieving in a large map, the \textit{AtexRace} do not record all function pairs observed in previously executions. One way to optimize the performance
is to only keep the recently frequently observed function pairs. Given an execution \( e \) and a number \( n (n \geq 1) \), we define a function pair \( <f_x, f_y> \) to be \( n \)-frequency with respect to execution \( e \) if \( <f_x, f_y> \) is observed in current and all the \( n - 1 \) previous executions. Specially, when the value of \( n \) is 1, the 1-frequency function pairs are those observed in the current execution. By keeping only, the \( n \)-frequency function pairs, the recorded function pairs are those frequently executed. Actually, it is reasonable not to sample these frequent function pairs to reduce sampling overhead. As a result, for each execution, the number of function pairs taken as input is small and does not increase with increasing number of executions.

But still, compare to the all-frequency, \( n \)-frequency may still miss some sampling information. With the optimization about the sampling rate and the process of race detection, it is possible to use all-frequency on AtexRacePlus due to its lower overhead than AtexRace. However, the size of history may still increase with the times of detection, which may bring the large overhead back, so we treat \( n \)-frequency function pairs as an optimization option of AtexRacePlus. The performance of \( n \)-frequency function pairs is also evaluated in the next section.

2.6 Evaluation

This section presents the evaluation on AtexRacePlus. We compared it with LiteRace and FastTrack and our preliminary technique AtexRace. Because FastTrack is one of the fastest and most widely used tools in this category. It fully detects data races and can be considered as a sampling tool with a rate of 100%. And LiteRace is the state-of-the-art in-house sampling tool. Both of them are representative and well known. And AtexRacePlus is based on AtexRace.
2.6.1 Implementation

We have implemented *AtexRace*, *AtexRacePlus*, FastTrack and *LiteRace* on top of *Pintool* [25], [30], a widely used binary instrumentation framework. Our implementation targets on multi-threaded programs with *Pthread* library on Linux 32 system. Note that, *Pintool* runs like a virtual machine [25] and incurs large overhead. A better implementation can be done as the original *LiteRace* implementation [16] (i.e., to integrate sampling tools into the program under testing at compilation time).

On Linux platform, *Pintool* modes each program as image that contains sections and each section consists of multiple routines (or functions). And one routine includes several basic blocks. Basic block is built instructions. Based on our testing, sampling on basic block level will bring a huge overhead that is even larger than full sampling, to say nothing of sampling on instruction level. So, we use a 32-bit integer to encode on the routine level. The first 6 bits are used as the image identifier and the remaining 26 bits are used as routines identifier per image. Totally, this encoding allows at most images and routines in each image, which is enough in practice. Note that, since the memory address of the object may be different in each execution, the image ID is matched with its name, and the routine’s ID is the offset between its address and its image’s address. In this way, the ID number are guaranteed the same during across execution.

Before each function invocation, an analysis routine is inserted. The analysis routine records all the function calls and produces observed function pairs and maintains the sampling flag of each
function pair. Besides, another analysis routine is inserted after each memory access operations. The analysis routine performs shadow memory operation and HBR pattern matching. Before performing those operations, the analysis routine will check the sampling flag. If the sampling flag is not turned on, the analysis routine will return without performing those operations. *Pintool* provides some static analysis API functions (i.e. INS *IsStackRead*(), to check if the memory access is a stack read access). Before instrumenting the program, *AtexRacePlus* performs static analysis on the program to calculate the sampling rate for each function with the Algorithm 3.

### 2.6.2 Benchmarks

We choose the *Parsec* benchmark suite 3.1 [26] to evaluate the race detectors. The suite consists of 13 benchmarks. After eliminating the benchmarks that are not multi-threaded or cannot be compiled under the Pin environment, we obtain seven benchmarks: *Blackscholes, Bodytrack, Canneal, Freqmine, Vips, Raytrace* and *Streamcluster*. In our experiments, we run each benchmark from *Parsec* for 100 times to collect their results. Table 2.2 gives the source code size (SLOC) of the eight benchmarks. It can be observed that the lines of code range from 1.3K to 246K. To further evaluate the performance of *AtexRacePlus*, we select the MySQL database server (v6.0.4), a widely used real-world program. The version we use, *mysql-6.0.4-alpha*, has 1,114,980 lines of code. Among the 399 test cases that comes with its distribution, 223 of them can be successfully executed in the Pin environment. We run all the 223 test cases in our experiment.
2.6.3 Experiment Setup

Our experiments were performed on a workstation (ThinkStation E32) with an i7-4770 CPU (eight cores), 16G memory, and 1T HDD. The workstation was installed with Ubuntu 12.04 x86 system. For AtexRace, we set its sampling rate and the value n (determining n-frequency function pairs) to be 10/100 and 2, respectively. For AtexRacePlus, we set the sample rate upper bound to 10/100. To compare with AtexRace we also performed experiment on two versions of AtexRacePlus. The first one is the AtexRacePlus with 2. frequency function pairs 2. The other one stores complete function pair history. For LiteRace, we adopt the fixed thread-local sampling configuration as defined in previous work [16].

2.6.4 Evaluation of Efficiency

For all techniques, Table 2.2 gives the time of the executions spent by Pintool and the seven tools of the benchmarks Table 2.3 shows the overhead of the race detectors compared to the time consumed by Pin framework. And it also reports the number of unique races (i.e., the number of variables in the source code) detected by each tool.

As expected, both LiteRace, AtexRace, AtexRacePlus and AtexRacePlus (2-frequency) are much faster than FastTrack. It can also be observed that LiteRace and AtexRace incurred almost the same average overhead excludes vips data, because it is not sensitive to our strategy which is proved in figure 2.8. On race detection capability, both LiteRace and AtexRace outperform FastTrack. At first glance, the results are surprising. However, it is known that sampling perturbs thread scheduling so a race detector with sampling runs different executions with the one without
sampling, even under the same test case. Such phenomenon is previously observed [31]. Table 2.3 shows that LiteRace detects 58% more unique races than FastTrack, all of the additional races are from the single benchmark Freqmine. AtexRace detects 19% more unique races than LiteRace. The above results indicate that AtexRace detect the greatest number of races except AtexRacePlus at a cost almost the same as LiteRace. Since these relatively small benchmarks do not give a doubtless evaluation of AtexRace, we further evaluate AtexRace on a large real-world database server MySQL. But before we present our empirical study on MySQL, we use Parsec to illustrate the advantage of cross-execution sampling of AtexRace.

One of key features of AtexRace is its cross-execution sampling, which may result in lower overhead with increasing number of executions.

The overhead on the $i$-th execution is calculated by the following formula:

$$\text{Overhead}_{\text{tool}}(i) = \frac{T_{\text{tool}}(i) - T_{\text{prog}}(i)}{T_{\text{prog}}(i)} \times 100\%$$

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>SLOC</th>
<th>PIN</th>
<th>FT</th>
<th>LR</th>
<th>TIME (Seconds)</th>
<th>AR</th>
<th>ARP</th>
<th>ARP (2- frequency)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackholes</td>
<td>1,380</td>
<td>230.98</td>
<td>1,626.37</td>
<td>1,299.24</td>
<td>1,170.02</td>
<td>1,255.97</td>
<td>1,267.39</td>
<td></td>
</tr>
<tr>
<td>Bodytrack</td>
<td>16,479</td>
<td>529.01</td>
<td>10,068.13</td>
<td>7,815.28</td>
<td>8,636.59</td>
<td>3,269.51</td>
<td>4,141.61</td>
<td></td>
</tr>
<tr>
<td>Carneal</td>
<td>2,847</td>
<td>747.15</td>
<td>6,029.38</td>
<td>3,286.47</td>
<td>4,391.34</td>
<td>3,512.30</td>
<td>3,437.08</td>
<td></td>
</tr>
<tr>
<td>Freqmine</td>
<td>2,192</td>
<td>1,320.72</td>
<td>53,878.72</td>
<td>60,544.95</td>
<td>62,448.83</td>
<td>38,803.10</td>
<td>32,204.54</td>
<td></td>
</tr>
<tr>
<td>Raytrace</td>
<td>14,867</td>
<td>2,193.70</td>
<td>42,294.85</td>
<td>23,129.40</td>
<td>29,187.86</td>
<td>10,715.54</td>
<td>10,729.29</td>
<td></td>
</tr>
<tr>
<td>Streamcluster</td>
<td>1,795</td>
<td>1,061.50</td>
<td>2,888.58</td>
<td>2,160.95</td>
<td>2,763.50</td>
<td>2,163.72</td>
<td>2,050.12</td>
<td></td>
</tr>
<tr>
<td>Vips</td>
<td>246,119</td>
<td>736.61</td>
<td>51,807.16</td>
<td>30,929.79</td>
<td>43,494.46</td>
<td>13,608.09</td>
<td>13,583.28</td>
<td></td>
</tr>
</tbody>
</table>

where $T_{\text{tool}}(i)$ represents the execution time under a tool on the $i$-th execution, and $T_{\text{prog}}(i)$
represents the native program execution time under Pin. We use the overhead of Fasttrack as the baseline to show the speedup of LiteRace, AtexRace, AtexRacePlus and AtextRace (2-frequency). The speedup is calculated using the following formula:

\[
\text{Quotient}_{tool}(i) = \frac{\text{Overhead}_{tool}(i)}{\text{Overhead}_{Fasttrack}(i)} \times 100\%
\]

Figure 2.8 uses linear regression to compare different tools on the overhead quotient(y-axis) over Fasttrack of each execution(x-axis). The lower value in y-axis indicates higher improvement in overhead reduction. From figure 2.8, we see that, overall, all the y-axis values are lower than 1. It proofs all the sampling technique reduces the time cost of dynamic data race detection. LiteRace incur almost the same overhead across executions (i.e., nearly a horizontal line). For AtexRace, overhead decreases with increasing number of executions, although the trend is less obvious in Streamcluster. It can also be observed that, with increasing number of executions, AtexRace’s performance becomes the faster than LiteRace on most of the benchmarks.

At this moment, regardless the MySQL’s result, AtexRace seems has more benefits on across sampling and race detection accuracy, while LiteRace will have a better performance on time consuming of the single testing. But when AtexRacePlus introduced, the things change to another story. Compare to AtexRace, it has a significant improving on time consuming, which only takes no more than half time of AtexRace taking. This amazing low overhead leads AtexRacePlus 43% faster than LiteRace. At the same time, compare to both LiteRace and AtexRace, it is excited us that AtexRacePlus even detected the most races. As we mentioned above, withing the increasing
of history, the data Retrieving overhead may also grow sharply, so to make the map history maintain in a steady size, we keep the 2. frequency as an optional choice. From Table 3, we can see $AtexRacePlus$ (2-frequency) is the fastest of all, this credits to the light map history as $AtexRace$’s and having all the optimizations of $AtexRacePlus$. However, the fewer number of races of $AtexRacePlus$ (2-frequency) caught our attention, but after further observing, we found almost all the missing races are come from $Frequmine$ and it is still very effective for others, which make this sampling method still acceptable.

Table 2.3 Overhead and number of unique data race in Parsec

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>FT</th>
<th>LR</th>
<th>AR</th>
<th>ARP</th>
<th>ARP (2-frequency)</th>
<th>FT</th>
<th>LR</th>
<th>AR</th>
<th>ARP</th>
<th>ARP (2-frequency)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackholes</td>
<td>6.04</td>
<td>4.62</td>
<td>4.07</td>
<td>4.44</td>
<td>4.49</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bodytrack</td>
<td>18.03</td>
<td>13.77</td>
<td>15.33</td>
<td>5.18</td>
<td>6.83</td>
<td>3</td>
<td>14</td>
<td>26</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Canneal</td>
<td>7.07</td>
<td>3.40</td>
<td>4.88</td>
<td>3.70</td>
<td>3.60</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Frequmine</td>
<td>37.79</td>
<td>44.84</td>
<td>46.28</td>
<td>28.38</td>
<td>23.38</td>
<td>115</td>
<td>192</td>
<td>221</td>
<td>250</td>
<td>156</td>
</tr>
<tr>
<td>Raytrace</td>
<td>18.26</td>
<td>9.53</td>
<td>12.29</td>
<td>3.88</td>
<td>3.89</td>
<td>13</td>
<td>12</td>
<td>21</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>1.72</td>
<td>1.04</td>
<td>1.60</td>
<td>1.03</td>
<td>0.93</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Vips</td>
<td>69.33</td>
<td>40.99</td>
<td>58.05</td>
<td>17.47</td>
<td>17.44</td>
<td>32</td>
<td>44</td>
<td>52</td>
<td>43</td>
<td>43</td>
</tr>
<tr>
<td>Sum:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 2.8 The linear regression of overhead quotient for different sampling techniques with increasing executions on Parsec benchmark applications
2.6.5 Result Analysis on MySQL

MySQL is a real-world program with more than one million lines of code. We run it against 223 test cases in the default order of the test script “mysql-test-run”. To eliminate the randomness, for each tool we repeatedly execute the test script 5 times. All the data reported is the average value of 5 repeated experiments.

Figure 2.9 depicts how the overhead quotient to Fasttrack (y-axis) changes across 223 executions (x-axis). Unlike benchmarks from Parsec where all repeated executions are conducted against the same test cases, each of the 223 MySQL executions is conducted against a different test case. Therefore, on MySQL, FastTrack (as well and LiteRace, AtexRace and AtexRacePlus) may incur different overhead on different executions. The formula to calculate the cumulative speedup to Fasttrack of the first \( i \) executions is the same as that on Parsec (i.e., Eq. 1). The results shown in figure 2.9 are as expected, AtexRacePlus and AtexRacePlus (2-frequency) incurs a lot lower overhead than other tools.

As for the AtexRacePlus 2-frequency, it does not record all observed function pairs but only keeps recently observed ones to avoid potentially unlimited increase on the number of function pairs. Figure 2.10 shows a comparison on the cumulative number of function pairs (y-axis) with the increasing number of executions (up to 223). The two lines represent the data by recording all observed ones (“All Pairs”) and recording recently observed ones (“2-frequency Pairs”), respectively.
It can be observed that, with increasing number of executions, the number of all function pairs also increases. After 223 executions, the number of observed function pairs is almost 80,000. If we keep all these function pairs, a large overhead on querying is inevitable, which may eventually offset the benefit of sampling. The speedup data shows the 2-frequency can further reduce the time cost.
2.6.6 Evaluation of Accuracy

Figure 2.11 gives the number of unique races that are detected by FastTrack, LiteRace, AtexRace, AtexRacePlus and AtexRacePlus (2-frequency) after 223 executions of MySQL. Table 2.3 shows the number of unique data races detected in 100 executions of each application in Parsec. Not surprisingly, compared with LiteRace, AtexRace, AtexRacePlus and AtexRacePlus (2-frequency) detect more unique races. What we have not expected is that our tools detect even more unique races than FastTrack. This is possible because sampling perturbs thread scheduling. As for the AtexRacePlus (2-frequency), it detects less unique data races than AtexRacePlus. Even the loss of history awareness will help to reduce the overhead, but it also decreases the ability of data race detecting. So, we decide the 2-frequency method can be an option provided to programmers.
2.7 Discussion

*AtexRacePlus* is proposed to detect races across executions. However, on a limited number of executions, *AtexRace* may initially incur larger overhead than that by FastTrack and *LiteRace*. Hence, *AtexRace* may not be a first choice used in single executions, but *AtexRacePlus* could be the one, because it has lower overhead and higher accuracy. Just like *AtexRace*, *AtexRacePlus* still has ability to reduce the overhead with the increasing number of executions, and not lower the race detect ability. Since *AtexRacePlus* records all the function pairs in the previous executions, it has complete history-awareness. It samples the data race detection based on the entire history of executions. It lowers the chances of same race being repeatedly reported.

Considering all the experiments result, it confirms that *AtexRacePlus* can be a replacement of other tools. It detects the most races with the smallest overhead.
2.8 Related Works

Data races [1], [12] are extremely difficult to be found and reproduced. Techniques can be classified into two categories: static and dynamic. Dynamic techniques detect data races at runtime by observing the monitoring the memory accesses. Dynamic ones analyses concrete executions to detect data races according to some rules (e.g., the lockset discipline [10], [32], [33] and the happens-before relation [1], [34], [35], [36], [37], [38]). Static ones [8], [9] can detect data race by analyzing the source code of a whole program. Both static techniques [6], [7], [8], [9] and dynamic techniques [1], [10], [13], [14], [35] aim to detect data races. Due to lack of runtime information, static approaches can easily report many false positives. Although dynamic techniques are relatively precise, they incur heavy overhead. Sampling techniques aim to lower the overhead of data race detection.

Many sampling approaches have been proposed on data race detection. CRSampler [22] also targets on sampling but its main purpose is at user site. It is based on hardware breakpoints and clock races to detect data races; where DataCollider [23] purely relies on hardware breakpoints to detect those occurred data race by suspending threads. LiteRace aims to sample memory accesses to reduce runtime overhead at developer sites, which is also the focus of this paper. Unlike LiteRace, our work AtexRacePlus aims at sampling by considering whether function pair being executed is already sampled.

CCI [39] proposes cross-thread sampling strategies to find causes of concurrency bugs based
on randomized sampling. Unlike race sampling techniques (e.g., CRSampler, DataCollider, PacerSZ, and LiteRace), CCI focuses on failure diagnosis. However, CCI may cause heavy overhead (e.g., up to 900% [39]) although it targets on lightweight sampling. Carisma [17] improves Pacer by further sampling memory locations allocated at the same program location for Java. Valor [40] infers data races by detecting region conflict, which has good performance compared with FastTrack.

Another branch of works aims to firstly predict a set of potential data races and then to verify them. RVPredict [41] achieves a strictly higher coverage than HBR based detectors. It firstly predicts a set of potential races and then relies on a number of production executions to check against each predicted race. Racageddon [31] aims to solve races that could be predicted in one execution but require different inputs. It still needs a larger number of executions to check against each predicted race [42], [43]. Both RVPredict and Racageddon have to solve scheduling constraints for each predicted race, which may fail. RaceMob [19] statically detects data race warnings and distributes them to a large number of users to validate real races. In such a run, the schedules are guided by the set of data race warnings to trigger real data races. This kind of approach is able to confirm real races but cannot eliminate false positives. Besides, it may miss real races if such races are not predicted in the (static) prediction phase.

DrFinder [12] tries to predict the happens-before relation to further expose races hidden by the happens-before relation. It dynamically predicts and tries to reverse happens-before-relations from observed executions. However, its active scheduling is also heavy (e.g., about 400% [12] for Java programs). The purpose of data race detection to find concurrent bugs because data races are
the main source of concurrent bugs [44].

To explore all possible executions is another direction to find concurrency bugs (e.g., Model checking [45], [46]). However, it is usually impossible to explore all the interleaving although they may achieve certain coverage [47]. Practically, enumerating each schedule is not practical for large-scale real-world programs, even with reduction techniques [48].

Therefore, to explore a small portion of interleaving space that are error prone is also one direction. Chess [46] sets a heuristic bound on the number of pre-emption to explore the schedules. Also, although systematic approaches avoid executing previously explored schedules, they usually incur large overheads and fail to scale up to handle long running programs. For example, Maple [49] is a coverage-driven [50], [51] tool to mine thread interleaving so as to expose unknown concurrency bugs. PCT [52], [53] randomly schedules a program to expose concurrency bugs, which also requires large number of executions. However, it is difficult to apply these techniques to large-scale programs such as MySQL.

Besides multi-threaded programs, data race may also exist in other kinds of programs, such even-driven programs such as android applications [54], [55], [56], concurrent library invocations [57], and modified program codes [58]. AtexRacePlus could also be adapted to detect these races. We leave it to future work.

2.9 Conclusion

We have proposed a new comprehensive sampling approach to achieve both high race detection rate and high efficiency. By adopting several novel designs, our prototype AtexRacePlus
can be a replacement of Fasttrack and LiteRace. This is confirmed by the experiments with benchmarks obtained from both Parsec benchmark suite and a real-world large-scale MySQL database.
References


[27] X. Xie, J. Xue, and J. Zhang, “Acculock: Accurate and efficient detection of data races,”


CHAPTER 3

DEEP DEFENCE BY EMBEDDING INVISIBLE WATERMARK

This chapter presents the contribution: Deepfake Defense by Embedding Invisible Watermark.

3.1 Summary

Deep learning technique has a significant improving in last years. Some deep neural networks (DNNs) based free tools have been released to create face swapping pictures and videos, which called” Deepfake”. Due to its believable and realistic characteristics, Deepfake techniques could bring us a great convenience for several areas such as education, art, and entertainment. While it was also found be used in generating revenge porn, fake news, economic fraud, which also bring us a lot of troubles.

To manage the usage of Deepfake techniques, a lot of Deepfake detection/defense tools have be proposed. To detect the distortion and biometric violations is the major direction for Deepfake detection, but most of them have lost their effectiveness with the evolution of Deepfake techniques. And to monitor the consist of frames is another branch, but this kind of approaches only work for videos. And the biggest problem is there is no released tool could have 100% effectiveness for Deepfake attacking, even their initial detection performance is very well, but the developing of Deepfake techniques will decrease their efficiency.
In this chapter we proposed an approach to defend Deepfake attacking by embedding invisible watermark which will never out of date. The method is based on Discrete Cosine Transform (DCT) and Quantization Index Modulation (QIM), besides, the watermark is reduced into binary format to represent embedding information. And the experiment shows our approach has enough power to defend Deepfake attacking in most picture format and major video format and the detecting accuracy is 100% and never out of date.

3.2 Introduction

Deep learning technique has a significant improving in last years. Some deep neural networks (DNNs) based free tools have been released to create face swapping pictures and videos, which called” Deepfake”, because it is the product of deep learning technique and fake. Due to its believable and realistic characteristics, Deepfake techniques could bring us a great convenience for several areas such as education, art, and entertainment. While it was also found be used in generating revenge porn, fake news, economic fraud [1] [2] [3], which also bring us a lot of troubles.

The deep learning could generate fake videos or images easily and cheaply, which means it could produce a large number of fake materials at low cost in a short time. figure 3.1 shows a result of Deepfake attacking. Ignoring the tag on the left-top corner, it’s very hard to distinguish which one is the real one by human eyes. To generate such a lifelike fake video may just need one hours to one day to training the neural network model with modern hardware equipment. After training process, a huge number of fake products could be generated just in several minutes.
Because of Deepfake’s simple principle, a lot of free Deepfake tools have been released, such as FaceApp, Deepfacelab, Zao. Their perfect performance and usability helped Deepfake spread fast. However, if it was used by someone with mischief maybe bring us a big trouble, and to eliminate such bad behaviors may be a privilege we need to pay for. With the helping of the fake videos, the scandals could spread faster than ever. For some important events, like present election, any rumor could lead to a totally different result, let alone with a realistic fake video. Because of lacking necessary knowledge for most masses, it is extremely expensive to refute the rumor. Manipulation of Deepfake defense is urgent.

Fortunately, there are lots of works been released which are focusing on Deepfake detection or defense. Detection based on biological characteristics is a popular area in early Deepfake defense area. One of them is based on eye blinking. The theoretical basis of this approach is human
eye blinking has strongly temporal correlation [4], and by building a Long-term Recurrent Convolutional Neural Networks (LRCN) [5] to compare current state to their previous states for distinguishing the open and close states. Another scheme is focus on detecting the inconsistent head poses [6]. This project claims the face swapping algorithm by neural network cannot guarantee to consistent the facial landmarks same as the original face.

Besides, there are also some achievements detecting the fake videos/pictures by the defects of Deepfake attacking algorithm. Based on M. Koopman’s research [7], the pictures attacked by Deepfake may bring the difference to photo response non uniformity (PRNU) [8]. While Y.Li’s method clammed Deepfake algorithm can only synthesize a fixed size of face images, in order to matching with the configuration of the sources face, it is necessary to undergo an affine warping. This warping operation leaves clues to distinguish from the original face [9]. comparing the face areas and their surrounding regions with a dedicated Convolutional Neural Network (CNN) model would help to detect the Deepfake attacking.


With the developing of the Deepfake technique, its defects and the vulnerabilities of biological characteristics have been fixed by the newer version of Deepfake applications. Additionally, the images are also a main target field of Deepfake attacking, but the schemes based on frames relationship won’t work on fake images very well. Besides, none of above approached could have a 100% accuracy detection and evolution of Deepfake techniques will decrease the accuracy of these tools. So, we proposed a method for Deepfake defense by embedding invisible watermark
which has a 100% accuracy and will be never out of date.

Invisible watermark embedding is a promising technique to protect the copyright when some works was been created. Once we applied it into Deepfake defense area, its simple implementation and 100% accuracy excited us. In order to defend Deepfake attacking, we modified the embedding algorithm. To make it practical, we have solved several challenges. Firstly, to make the watermark invisible which means it’s an information could not be observed by human eyes even with the high magnification. Secondly, embedding and extracting the watermark successfully are our next challenge. Lastly, the solution should be robust rough to against major attacking, like cropping, rotation or JPEG compression.

Our solution is embedding the watermark into the coefficient of DCT by the variant of QIM algorithm Dither Modulation. First, the source image will be transformed from RGB to YCbCr, and only Y channel will be used to embedding the information. Second, the watermark will be created by cropping the human face in the source image, then it will be expanded into the same size as the Y channel and converted into binary format. Third, DCT will be applied to 8*8 pixels block of Y channel. The next step is to embed the watermark information into the coefficients of each pixel’s block. Finally, inverse DCT algorithm will be applied to the Y channel, and recomposed with Cb and Cr channels, then it is also necessary to change it back to RGB format. Then the embedding process has completed. And the extract process is the opposite operations of embedding process. If the defense target is a video, an additional pretreatment and post-treatment should be applied: splitting it into frames before embedding process and reassembling the embedded frames into new video after the embedding process are necessary. As for the extracting
part, the video will be split into frames again, and extract the watermark from the frames.

We have implemented this scheme and evaluated it on 7 videos, as well as the JPEG format of the videos’ frames. The experimental results surprisingly show that the accuracy of detection is 100% and bring no extra overhead.

Rest of the paper is showed as follows: Section 2 talks some background about Deepfake and watermark embedding information, section 3 is about our motivation, section 4 will describe the algorithm of our scheme, in section 5, we will show the experiments’ result and analyze, section 6 is talking about some related works and future work, and the conclusion will be discussed in section 7.

3.3 Background

3.3.1 Deepfakes

Even the effect of Deepfake attacking looks very lifelike, but its principle is quite simple, especially with the help of Deep Learning. Basically, it could be split into two part: training process and repair process. figure 3.2 shows the details of the training process. During this process, the input to the training model could be thousands of frames of a video or images of person 1. Then encoder and decoder (neural network) will be applied to grep the features of person 1. After several times, the neural network model will have the features of person 1. The result will be improved withing the increasing of the training times. After obtained the features, then it moves to repair process. Showed as figure 3.3. Once the model received a new input of person 2, it will treat the
new input as person 1 with incorrect features, and the model will repair it with person 1’s features which were gathered during training process. After all the features applied to this new input, the neural network model will output a fake frame or image of person 1 with person 2’s expression. Just like figure 3.1 (b) showed.

![Training process](image1)

**Figure 3.2 Training process**

![Repair process](image2)

**Figure 3.3 Repair process**

3.3.2 Videos and Pictures Color Space

In order to represent colorful image information, a mathematics model color space has been introduced, which usually has three- or four-color channels. In modern society, the color space
models have been widely used for various areas, such as computer graphics, image processing, TV broadcasting, and computer vision.

RGB is the most popular model to store the color information, once a picture was loaded, each pixel will have three values range from 0 to 255, which stand for the shade of three base color: red, green, blue (in some format, the pixel will have 4 values, compare to the 3 values model, it has an extra channel stands for the transparency). Since human eyes are more sensitive to luminance than chrominance, and most of vision lossy compression algorithm will let the chrominance channel loss more information than luminance channel. Based on above theory, it seems to embed the watermark into the luminance channel could improve the invisibility and robust. So RGB model is not a perfect model, because it mixed of color and intensity information and its non-uniform characteristics [12]. So YCbCr (YUV) has been introduced where Y channel is the luminance and Cb, Cr channels (also called U and V channels) stand for chrominance. RGB and YCbCr can be transformed reversible by below equations:

\[
\begin{pmatrix}
  Y \\
  Cb \\
  Cr
\end{pmatrix} = \begin{bmatrix}
  16 \\
  128 \\
  128
\end{bmatrix} + \begin{bmatrix}
  0.279 & 0.504 & 0.098 \\
  -0.148 & -0.291 & 0.439 \\
  0.439 & -0.368 & -0.071
\end{bmatrix} \begin{bmatrix}
  R \\
  G \\
  B
\end{bmatrix}
\] (1)

\[
\begin{bmatrix}
  R \\
  G \\
  B
\end{bmatrix} = \begin{bmatrix}
  1.164 & 0.000 & 1.596 \\
  1.164 & -0.392 & -0.813 \\
  1.164 & 2.017 & 0.000
\end{bmatrix} \begin{bmatrix}
  Y - 16 \\
  Cb - 128 \\
  Cr - 128
\end{bmatrix}
\] (2)

R, G, B in the equation (1) [12] and equation (2) [12] are the values of red, green, yellow for each pixel. While the Y, Cb, Cr are the values of Y, Cb, Cr channel.
3.3.3 Discrete Cosine Transform

Discrete Cosine Transform (DCT) is a widely used transformation technique in signal processing and data compression, which was first introduced by Nasir Ahmed [13]. DCT is a separable Fourier-related transform, and its core transformation is Cosine function. Besides the orthogonal transformation property, the base vector of DCT transformation matrix could also represent the features of human voices’ signals and image signals. For example, the JPEG compression’s core part is DCT. For 2-Dimensional matrix, DCT is expressed by the equation (3), and equation (4) is the inverse transform of DCT.

\[
F(u, v) = a(u) a(v) \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} f(i, j) \cos \left( \frac{(2i + 1)u\pi}{2N} \right) \cos \left( \frac{(2j + 1)v\pi}{2N} \right)
\]

\[
a(\lambda) = \begin{cases} 
\frac{1}{\sqrt{2}}, & \lambda = 0 \\
1, & \lambda > 0 
\end{cases}
\]

\[
f(i, j) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} a(u) a(v) F(u, v) \cos \left( \frac{(2i + 1)u\pi}{2N} \right) \cos \left( \frac{(2j + 1)v\pi}{2N} \right)
\]

\[
a(\lambda) = \begin{cases} 
\frac{1}{\sqrt{2}}, & \lambda = 0 \\
1, & \lambda > 0 
\end{cases}
\]

Figure 3.4 is a sample arrangement DCT result of an 8*8 block. Figure3.4 (a) is the original values of a grayscale 8*8-pixel block. Figure 3.4 (b) is its result after DCT operation. The DCT coefficients are ordered as zigzag scan and 4 types of frequency brands (DC, low, mid, high) from
the left-top corner to right-bottom corner. And the human eyes are more sensitive to the lower brand.

<table>
<thead>
<tr>
<th></th>
<th>78</th>
<th>70</th>
<th>82</th>
<th>79</th>
<th>73</th>
<th>77</th>
<th>68</th>
<th>71</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>67</td>
<td>83</td>
<td>86</td>
<td>85</td>
<td>69</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>75</td>
<td>87</td>
<td>81</td>
<td>70</td>
<td>77</td>
<td>85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>79</td>
<td>78</td>
<td>66</td>
<td>56</td>
<td>68</td>
<td>89</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>67</td>
<td>59</td>
<td>62</td>
<td>72</td>
<td>86</td>
<td>94</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>61</td>
<td>59</td>
<td>72</td>
<td>85</td>
<td>94</td>
<td>96</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>71</td>
<td>75</td>
<td>87</td>
<td>91</td>
<td>93</td>
<td>97</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>82</td>
<td>83</td>
<td>97</td>
<td>104</td>
<td>102</td>
<td>98</td>
<td>71</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>629</th>
<th>-33</th>
<th>-8.81</th>
<th>9.7239</th>
<th>-7</th>
<th>11.836</th>
<th>0.9431</th>
<th>0.1256</th>
</tr>
</thead>
<tbody>
<tr>
<td>-24.334</td>
<td>30.081</td>
<td>12.075</td>
<td>-19.86</td>
<td>7.7076</td>
<td>-0.446</td>
<td>17.935</td>
<td>-0.552</td>
<td></td>
</tr>
<tr>
<td>16.1543</td>
<td>15.75</td>
<td>-44.77</td>
<td>6.8666</td>
<td>2E-14</td>
<td>-0.371</td>
<td>0.3536</td>
<td>-0.416</td>
<td></td>
</tr>
<tr>
<td>-19.712</td>
<td>-14.96</td>
<td>-0.135</td>
<td>17.675</td>
<td>0.3359</td>
<td>0.042</td>
<td>-0.428</td>
<td>-0.301</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5.9519</td>
<td>22.192</td>
<td>-0.258</td>
<td>-4E-15</td>
<td>0.0769</td>
<td>0.008</td>
<td>-0.047</td>
<td></td>
</tr>
<tr>
<td>-0.0232</td>
<td>-0.452</td>
<td>16.779</td>
<td>-20</td>
<td>-0.252</td>
<td>0.3148</td>
<td>0.1463</td>
<td>-0.538</td>
<td></td>
</tr>
<tr>
<td>-0.197</td>
<td>0.4202</td>
<td>0.3536</td>
<td>0.0678</td>
<td>1E-14</td>
<td>0.1522</td>
<td>-0.226</td>
<td>-0.102</td>
<td></td>
</tr>
<tr>
<td>-0.1829</td>
<td>0.4018</td>
<td>-0.481</td>
<td>0.0215</td>
<td>0.408</td>
<td>-0.104</td>
<td>0.1269</td>
<td>-0.071</td>
<td></td>
</tr>
</tbody>
</table>

(a) (b)

Figure 3.4 Quantization Index Modulation (QIM)

3.3.4 Quantization Index Modulation (QIM)

In last years, there are many watermark embedding methods have been proposed including filters, Least Significant Bit (LSB) [14] and QIM [15].

The QIM algorithm was proposed by Brian Chen and Gregory W. Wornel, From the simulation experiments about the embedding methods, QIM reached a balance of embedding efficiency, embedding distortion and robustness. Based on the QIM method, they proposed Distortion-Compensated QIM, this method could make a good compensate to distortion.

The main idea of QIM is to select the different quantizers based on the embedding information, so that the embedded the signal will have the watermark’s information. The quantization function is represented by equation 5:
\[ f = \left[ \frac{I}{\Delta} \right] \cdot \Delta \]  

(5)

\( \Delta \) is quantization step, while the \([\ast]\) means rounding operation. \( I \) is the information need to be embedded.

From equation (5), it is very obvious that is a nonreversible many-to-one function. The output will be discrete values which means it has anti-interference ability in tolerance range. Equation (6) is a standard quantization table for luminance channel during JPEG compression process. Each element in the matrix is a quantization step. A larger value of a step will bring a more lossy during the quantization process. With the knowledge we obtained from last subsection, we could shorten the step for the lower bands area to reduce the distortion between the embedded image and the original one. As for other users, the modification on the matrix could be treated as a unique private key to extract the watermark.

\[
Q_Y = \begin{bmatrix}
16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\
12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\
14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\
14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\
18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\
24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\
49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\
72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \\
\end{bmatrix}
\]  

(6)
3.4 Motivations

3.4.1 Beneficial Use and Harmful Use of Deepfakes

Deepfake technique has been applied to many areas of human society once it been created. Specially in the areas of education, art, and the promotion of individual autonomy, just as Chesney, Bobby mentioned in 2019 [16]. For example, with the helping of Deepfake to create Avatar won’t spend 237 million again.

Like every technology, Deepfakes also could be used to cause a broad spectrum of serious harmful events [16]. It already be discovered a lot of revenge Deepfake videos and super stars’ Deepfake porn has been uploaded onto porn websites. And the fake video of former U.S. president Barack Obama has been widely spread on Youtube [17].

Based on above, the Deepfake technique just like a a double-edged sword depending on the one using it. Because of its realistic characteristic, it has brought us a big trouble to distinguish whether a picture or a video is real. Some means should be introduced to manage how to use Deepfake.

3.4.2 Limitations of Existing Approaches

There is a lot of Deepfake detection strategies have been proposed. Some of the detectors are based on the biological characteristics: eye blinking [4], head pose [6] and so on. Besides, some of others are focus on the defects of the Deepfake technique. M. Koopman A. Macarulla Rodriguez and Z. Geradts introduced a detection method based on photo response non uniformity (PRNU)
pattern [7] where PRNU was been introduced in 2006 [8]. They are every effective solution for the early version of Deepfake applications. But withing the evolution of the techniques, their false negative results are improving. The AdvIT [11] and David Guera’s method [10] to detect Deepfake attacking are based on the relationships between frames. This principle makes these kinds of approach could only work for videos rather than images. But images are also major target areas of Deepfake attacking.

In summary, useless for most recent Deepfake attacking, limitation of defense area, detection accuracy and watermark embedding robust are the motivations for us to work in this work.

3.5 Our Approach

3.5.1 Goal and Challenges

Our goal is to embed an invisible watermark into an image which we called the cover image. And we’d like to use the face in the image as the embedding watermark since almost all the Deepfake attacking targets are the human’s faces. So, the first challenge for us is the face recognition and cropping it as the watermark. The next tricky part is how to embed the watermark invisible that means the watermark could not be perceived by human eyes but a specific extracting algorithm. And the core challenge is the embedded information should have enough robustness to defend the major vision attacking, like cropping, rotation, format conversion and compression.

3.5.2 Process of Approach

1) **Face Recognition:** As our approach is defending Deepfake attacking, and the main target
of Deepfake attacking is human face of the cover image. We choose the face in the cover image as the watermark. So, the first step is using face recognition algorithm to find the face area, then cut the face out from the image. For face detection, 68 facial landmarks detection in DLib library which was proposed by Vahid Kazemi and Josephine Sullivan in 2014 [20] is applied in our approach because it is an algorithm to precisely estimate the position of facial landmarks in a computationally efficient way [20]. figure 3.5 [21] shows how these 68 landmarks layout on a image based on the human’s face features. The output would be a square picture within a face detected which will be treated as the watermark.

![Figure 3.5 68 facial landmarks layout](image)

2) **Watermark Process:** Since the watermark could be a colorful picture or a gray scale picture, it needs to be converted into binary format. Then expand the size of the watermark to the cover image size. Below are the processing details:

- Change the watermark’s from RGB model to YCbCr model, and only Y channel will be used, Cb,Cr channels will be abandoned. We can treat the Y channel as the gray-scale

90
watermark.

- In gray-scale image, each pixel only has one value in range from 0 to 255 indicates the illumination. Then all the values larger than or equal 127 will be modified to 255 and the values smaller than 127 will be changed to 0 since 127 is the mid-value between 0 and 255. The binary image obtained (0 is zero, 255 treats as one).
- w, h are the width and height of cover image. Then we expand the binary watermark to w, h by repeating the its values.

Figure 3.6 Embedding process

3) **Embedding Process**: If the cover source is video, the first step is the split it into frames and save the frames into any picture format. The cover image needs to be transformed from RGB to YCbCr. And Y channel is the only channel will be used to be embedded the watermark since it has the smallest lossy when compression operation applied. The next step is to split the Y channel
into 8x8 pixels block. After that Discrete Cosine Transform (DCT) will be applied to each pixel block. Each coefficient will be modified with the binary watermark information by Dither Modulation Quantization Index Modulation (QIM-DM) which is a variant of QIM. With the using of quantizer Q (*) equation (5), the embedding function is showed as equation (7)

\[
C'_k = Q(C_k + d(W_i)) - d(W_i) \quad (7)
\]

\[
d(W_i) = \begin{cases} 
  d0, & W_i = 0 \\
  d1, & W_i = 1 
\end{cases} \quad (8)
\]

\[
d0 = R \cdot \Delta, \quad R \in (0, 1] \quad (9)
\]

\[
d1 = \begin{cases} 
  d0 + \Delta/2, & d0 < 0 \\
  d0 - \Delta/2, & d0 \geq 0 
\end{cases} \quad (10)
\]

In equation (7), Q (*) is a quantizer equation (5), Ck is the Kth coefficient of an 8*8-pixel block of the cover image. Wi is the i\textsuperscript{th} vector of the watermark. While d (Wi) is defined by the value of Wi, equation (8) shows the details. R in Equation (9) is a random generator, and R and \Delta could be user’s private keys, only with this key, the watermark could be extracted. We defined d1 in equation (10). As for the quantizer step, to reduce the distortion of the embedded image, the step for each pixel of the 8*8 block has been shorten as Eq. (11). Below shows the shorten process:

- Do ten times shorter operation for each step,
- Round the values to their nearest integer,
- Make the DC, low and mid brands as 1 due to human’s eyes are really sensitive to these areas.
After above steps, we will get a new coefficient $C'_k$ which is showed in equation (7)’s left side. The next step is to do the inverse DCT process to each 8*8-pixel block, then we will get a new luminance channel $Y'$ which carried the watermark information. $Y_0$ will be reassembled with $C_b$ and $C_r$ channel and the final step is to return to RGB model. Now, the image with invisible watermark has been obtained and embedding process is completed. If the target is video, generate a new video by all the embedded frame is necessary. Figure 3.6 showed all the process for the embedding method.

4) **Extracting Process:** The extracting process is quite similar with the embedding process, but all the operations are inverse. Figure 3.7 showed the basic extracting steps:

- DCT will be applied to each non-overlapped 8*8-pixel block.
- Quantizer equation (5) will be reused during extracting process. Equation (12) and equation (13) will generate 2 locations $L_0$ and $L_1$
Next step is to compute the distance to $C'_k$ with last step’s two locations, just as equation

$$L_0 = Q(C'_k + d0) - d0 \quad (12)$$

$$L_1 = Q(C'_k + d1) - d1 \quad (13)$$

Figure 3.7 Extracting process
(14) and equation (15) showed:

\[ D_0 = |L_0 - C'_k| \]  
\[ D_1 = |L_1 - C'_k| \]  

(15)

As equation (16) showed, if \( D_0 > D_1 \), the watermark’s pixel value \( W_i \) will be assigned as 1, otherwise it will be 0. Then the binary watermark’s recovery is completed.

\[ W_i = \begin{cases} 
1, & D_0 > D_1 \\
0, & D_0 \leq D_1
\end{cases} \]  

(16)

5) **Deepfake Defending**: Since only the one that embedded the watermark obtained the private keys, others cannot extract the watermark successfully. And, if the extracted watermark is intact and matching with the face in the image, we could make the conclusion the image or video never been attacked, which means it is the real one. Otherwise, it is a fake one.

3.6 Experiments

3.6.1 Experimental Setup

We implement our defense method by using Python 3.8 with OpenCV and Numpy. Our implementation targets on major formats of images and videos. *DeepFaceLab* [22] will be used as attacking simulation.

All the experiments were tested on a PC (HP Pavillion All-in-One 24-xa0xxx) with the following features:

- CPU Intel Core i7-8700T @ 2.40 GHz (6 cores)
• Ram memory 16GB (DDR4, 2666MHz)
• Hard disk 250GB (SSD)
• Graphic Card Nvidia GeForce GTX1050 (4 GB)
• Windows 10 64-bits

The tested benchmarks are come from, the details are showed on Table.3.1.

<table>
<thead>
<tr>
<th>Video name</th>
<th>Resolution</th>
<th>Size</th>
<th>Duration</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black cloth man</td>
<td>3840*2169</td>
<td>7.11GB</td>
<td>00:07:04</td>
<td>MOV</td>
</tr>
<tr>
<td>Standing man</td>
<td>1920*1080</td>
<td>5.18MB</td>
<td>00:01:04</td>
<td>MP4</td>
</tr>
<tr>
<td>Deborah Birx</td>
<td>1920*1080</td>
<td>171.7MB</td>
<td>00:10:27</td>
<td>MP4</td>
</tr>
<tr>
<td>Mike</td>
<td>1920*1080</td>
<td>89.9MB</td>
<td>00:05:24</td>
<td>MP4</td>
</tr>
<tr>
<td>Donald</td>
<td>1920*1080</td>
<td>245.3MB</td>
<td>00:15:04</td>
<td>MP4</td>
</tr>
<tr>
<td>Speeching woman</td>
<td>1920*1080</td>
<td>81.7MB</td>
<td>00:09:01</td>
<td>MP4</td>
</tr>
<tr>
<td>Ivanka</td>
<td>1280*720</td>
<td>5.13MB</td>
<td>00:00:53</td>
<td>MP4</td>
</tr>
</tbody>
</table>

3.6.2 Result and Evaluation

We have tested on 7 videos (some are from YouTube [23] and some are made by our-self). And all the videos are attacked by DeepFakeLab. In order to evaluate the robustness, we also cut some frames of the videos and transform to JPEG format to simulate the JPEG compression attacking. To save space, we only present a sample result in this paper due to all the processing are similar. But we listed all the data of the 7 videos. The Figure 3.8 is the comparison between original frame and watermark embedded frame. Just as observed, the watermark is totally invisible, which means the attacker is hard to realize the defense method has been embedded. And We also evaluated our embedding method with the evaluation criteria: SSIM, PSNR and NCC, which are
three major indexes to evaluate the similarity of two picture.

Structural similarity (SSIM) index is a method for measuring the similarity between two images which was introduced by Zhou Wang in 2004 [24]. It was defined as equation (17):

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$  \hspace{1cm} (17)$$

Where:
• $\mu_x$ the average of $x$;

• $\mu_y$ the average of $y$;

• $\sigma^2_x$ the variance of $x$;

• $\sigma^2_y$ the variance of $y$;

• $\sigma^2_{xy}$ the co-variance of $x$ and $y$;

• $c_1 = (k_1L)^2$, $k_1 = 0.01$;

• $c_2 = (k_2L)^2$, $k_1 = 0.03$; • $L$ is a dynamic range of pixel values.

SSIM’s range is from 0 to 1. If it equaled to 1, it means the two images are exactly the same. And it is a such evaluation index that could better reflect the subjective feelings of the human eye. From Fig 9, all the SSIM of benchmarks are around 0.9, which means the human eye can hardly see the watermark.
PSNR is abbreviation of peak signal-to-noise ratio, which is widely used as a quality measurement between the original image $I$ and its variant $I'$ in decibels (dB). It reflects the ratio between the maximum possible power of a signal and the power of corrupting noise that affects it. Usually the range of PSNR will in $[20, 40]$, and the higher value of PSNR means the better quality of reconstructed image. Equation (18) shows the calculation detail of PSNR:

$$PSNR = 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE) \quad (18)$$

Where $MAX_I$ is the maximum possible of of the input signals, and it could be valued as 255 in our case. MSE is defined as equation (19):

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i, j) - I'(i, j)]^2 \quad (19)$$
Where $m$ and $m$ in equation (19) are the width and height of the images. $I(i, j)$ is the original image, and the $I'(i, j)$ is the image with embedded watermark.

NCC (normalized cross-correlation) is a quality metric concept to evaluate the robustness of watermark algorithms by comparing the difference between embedded watermark and the extracted watermark [25].

![Figure 3.10 PSNR](image)

The NCC of embedded binary watermark $w$ extracted binary watermark $w'$ is defined as equation (20) [25]:

$$
NCC(w, w') = \frac{\sum_{i}^{n} (w_i - \bar{w})(w'_i - \bar{w}')}{\sqrt{\sum_{i}^{n} (w_i - \bar{w})^2 \sum_{i}^{n} (w'_i - \bar{w}')^2}}
$$ (20)

Where $\bar{w}$ is the average value of $w$, while $\bar{w}'$ is the average of $w'$. The range of NCC is
[-1,1], If it equals 1, it means the two images are identical, if the value equals to -1, it means the two images are completely opposite, if it equals to 0, it means the two images are uncorrelated.

From figure 3.10 and figure 3.11, we can notice that these two evaluation methods do find the difference between the original image and the embedded image. But these two algorithms can’t reflect the subjective feelings of human eyes. And only with the original materials could it find the difference, but most Deepfake attacked resources will not provide the original version.

The comparison of attacked image and the original image could be found in figure 3.1. We extracted the watermark from the attacked frame and the not attacked frame with the private key. There are two main features shows the frame has been attacked. First, from a subjective perspective, it is very easy to notice that the face in the watermark is quite different from the one appears in the video. Based on our design, extracting a totally different face from the face inside the frame could sufficiently proof the video is fake. Second, from the technical level the DeepFake attacking will
only swap the face region, which means the watermark in the face area will be destroyed. In another word, the attacking tool changed the parameters of the face area, and it won’t be able to recover the watermark after the attacking. The result was showed in figure 3.12, compare to the face in figure 3.1 (b), the extracted face is definite from a different person. Also, the face region (showed as red square) has been destroyed. With the above two evidences, the figure 3.1 (b) could be proven to be fake.

Figure 3.12 Watermark extraction comparison
3.6.3 Discussion on Robustness

Our main idea is to embed an invisible watermark to defend the DeepFake attacking. The robustness is really important for our method since we do not want to lose the watermark when some modification applied, such as cropping. In figure 3.13, we did 3 major graphics attacking: cropping, rotation, JPEG compression, and the result shows that our embedding method have enough robustness to against the these attacking.

3.7 Related Work

The Deepfakes are extremely difficult to detect. No matter the techniques are based on
biological characteristics or other principles aim to detect the Deepfakes. The method based on biological characteristics [4] [6] claims Deepfake attacking will produce such a phenomenon that outline with the human physiological characteristics. The approach based on the vulnerability of Deepfake is the attacking will bring some unnatural transition at the junction of the face area and its surrounding parts [7] [9]. Above detection means are no longer effective since the evolution of Deepfake attacking. And the techniques to detect the inconsistent frames [10] [11] are more focus on the videos due to it’s very hard to find a relationship between different fake pictures.

Besides, during the watermark process, the watermark could be the author’s the unique biological characteristics, like fingerprint, finger vein or a hybrid of biological characteristics [26] [27] [28] to improve the security of the method.

Based on the result of Atexrace [29], we believe less sampling with optimization heuristics can lead to net benefit [29]. So, to make improve the efficiency, there is no need to detect all the frame of video, and all the area of pictures due to Deepfake attacking only aim on human face area, which will be treated as our future work.

3.8 Conclusion

We have proposed a new Deepfake defense method to achieve both 100% accuracy and robust. Our prototype shows its potential to defense major Deepfake attacking. This is confirmed by the experiments’ result. Besides, this efficient solution could be an upload standard for videos or pictures source website.
References


106

CHAPTER 4
CONCLUSIONS AND FUTURE WORK

The main goal of my dissertation was to make contribution to computer security. AtexRace and AtexRacePlus showed a new strategy on concurrent bugs’ detection. No matter on accuracy or on speed, AtexRacePlus shows outstanding ability, which makes it as the best sampling race detector. Additionally, the scheme that embedding invisible watermark to defend the Deepfake attack is simple and streamlined. Even new attacking techniques may appear, this defense method on source materials with private key will never out of date.

As for future work, I believe both AtexRacePlus and Deepfake defense method are still having potential for improvement. There are lots of races which are hidden in mobile application and concurrent libraries, which is a branch AtexRacePlus could be developed. And the sampling strategy of AtexRacePlus could be also applied to detect other current bugs, such as atomicity violation. As for the Deepfake area, the approach could be evolved by new embedding information, such as fingerprint, finger vein. Or if we could find a better embedding method, it also could help to improve the defense ability.