Dancing with Uncertainties: A meritorious case for race conditions and inconsistencies in operating systems

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
in
Computer Science and Engineering by Research

by

Bhanu Dev Chaluvadi
201102082
chaluvadibhanu.dev@research.iiit.ac.in

International Institute of Information Technology
(Deemed to be University)
Hyderabad - 500 032, INDIA
July 2024

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Dancing with Uncertainties: A meritorious case for race conditions and inconsistencies in operating systems" by Bhanu Dev Chaluvadi, has been carried out under my supervision and is not submitted elsewhere for a degree.

_____ _____

Date Adviser: Dr. Kannan Srinathan

To my parents

Acknowledgments

First and foremost, I cannot thank my professor Dr. Kannan Srinathan enough. He has always been an inspiration, guide, and someone I look up to. My appreciation for research and learning in general increased manifold after meeting him. Working closely on homomorphic encrycption with him for a few months is something I cherish.

I am thankful to my labmates for the collaboration and providing a nice working environment. I cannot not mention Durgesh Pandey, for giving me company, being a critique of my work and as a whole, being a nice labmate. Also, thanks to my lab seniors Ravi Kishore, Chiranjeevi, and friends Hardhik, Sunayana and Anupriya for being in the same lab and making it feel like we all are in a journey together.

College is about friends and memories, nothing of this work or stay in college would have been enjoyable without my dearest friends Kt, Sunil, Sumant and Ponna. On a similar note, cannot thank Pranav and other halo friends enough for being there for me after college. And to Guru and Sudhir for providing nice company and inspiration during my late years in research.

Last but not the least, nothing of this thesis or my endeavours in education would have been possible without the constant love and support from my wife Jeevana, parents and sister. They are always caring, supportive about my academic, physical and mental well-being.

Lastly, I would like to thank all the great researchers and teachers from times immemorial, including my own teachers. Students are a product of teachers, their dedication and passion towards teaching. I feel grateful to be taught by amazing teachers all through my life. I cannot quote everyone of my teachers here, but some of the teachers and courses I thouroughly enjoyed during my stay in IIIT are Data Structures, Complexity and Adv. alogrithms by Dr. Kishore Kothapally, Non-violence by Dr. Nanda Kishore Acharya and of course, almost all the courses by my professor Kannan Srinathan sir. They were all engaging, thought provoking and so much to learn. I would also like to thank Dr. Kamal Karnapalem for giving some really nice advice during HV in my first year. I thank the intitute for providing a nice environment which nurtured learning. Finally, though I am sure my work here is infinitesimally small in the grand scheme of things, I hope this work of mine helps me to embark more on the journey of research, in times to come.

Abstract

There are various uncertainties in real world operating systems. Race conditions, cache inconsistencies, unpredictable thread orderings are some examples of such undesirable phenomena. These cause unexplained bugs in software, cost millions to organizations and can lead to much bigger issues like a major power blackout or a fatal error in diagnostic machine. There is a vast amount of research to mitigate these and make the systems more predictable. What if we take an alternate look at these uncertainties?

Can we use these uncertainties to get something useful? Consider this problem of two millionaires. Two rich persons intend to find who is richer without knowing each other's wealth. They don't trust anyone else to do this comparision for them. Can we use the above uncertainties to solve this problem? This problem is formally known as the millionaire problem and is a well studied problem in cryptography. This is one instance of a family of problems named Secure Multiparty computation, where multiple parties compute a common function on their inputs without revealing anything about their own inputs other than what is revealed by the output of the function itself. This is a fundemental problem in cryptography, can be achieved using a construct named Oblivious transfer, also a fundemental and crtitical problem in the field of cryptography.

Cryptography has always used noise and randomness to achieve security, atleast theoritically. But the noise or uncertainty available in operating systems is never used to achieve security. There were some attempts, for example, generating random numbers from race conditions, but they were all limited to such simple usecases. In this thesis, we use the uncertainties in operating systems to achieve oblivious transfer and secure multi party computation. We propose two protocols to achieve these constructs, using different uncertain mechanisms in operating systems. We use CPU instruction order uncertainty, thread reordering, and race-conditions, in the construction of these protocols respectively. We implement them on various real world systems and prove that this novel and alternate view is indeed practical and can be used to create real world cryptographic protocols. We also propose and implement a library which, given any uncertain channel of certain characteristics, implements secure multiparty computation between parties.

Contents

List of Figures

Figure Page

List of Tables

Chapter 1

Introduction

Lets go back to 2003. There was a widespread power outage throughout parts of Northeastern and Midwestern United States. It affected an estimated 50 million people. This was caused by a software bug, more precisely, a race condition which stalled control room alarm system for over an hour. Now, lets look at Therac-25, a computer controlled radiation machine which gave its patients radiation doses that were hundreds of times greater than normal. This was also caused by concurrent programming errors. A more recent such example would be a glitch in NASDAQ's software during Facebook's IPO which resulted in a purported 500 million dollars in loss.

All the above outages are caused by concurrency errors like race conditions. These occur because of inherent uncertainties in operating systems. For example, the order of execution of different threads and processes is not completely predictable, an interrupt can occur at any time, a process could be preempted, scheduling algorithms vary and so on. There has been a lot of research into this area, specifically, to mitigate these concurrency issues. Some of these are in the field of identifying and reproducing concurrency bugs, some propose new framework/OS proposals to avoid such bugs, and so on [10] [12] [16] .

We take an alternate view of these uncertainties. We show that these uncertainties, under certain conditions can be useful for security, to the extent that they can be used to simulate fundemental cryptographic protocols like secure multi-party computation and oblivious transfer.

1.1 Secure multi party computation

Secure multi party computation is a protocol in which multiple parties try to securely compute a function on their private inputs. In other words, one party does not want any other party to know its input other than what is already revealed by the output of the function. This is a fundemental and important problem in cryptography.

1.2 1-2 Oblivious transfer

Oblivious transfer is a protocol in where a Sender S has two inputs a and b, receiver R has a bit i. Receiver likes to receive a if i = 0 and b if i = 1. S should be oblivious of the value of i. This protocol is complete for secure multiparty computation, i.e, given oblivious transfer, secure multiparty computation can be achieved without any additional primitives.

1.3 Contributions

Our contribution in this thesis is two-fold. Firstly, we put forth the novel idea (to the best of our knowledge) of achieving cryptographic protocols using uncertainties occurring in operating systems. Secondly, we model and propose multiple protocols for this, based on various mechanisms in operating systems. These protocols are implemented on different real world systems thus confirming they are practical and realistic. We also propose a library which can be used to implement secure computation given an instance of any uncertainty occurring in a real world system.

1.4 Organization of the thesis

Organization of the thesis is as follows. In chapter 2, my research journey at IIIT Hyderabad is presented briefly. In chapter 3, we present the main modelling we developed to achieve secure multiparty computation from uncertainties in operating systems. In chapters 4 and 5, we present the protocols from various uncertainties, their implementations and results. In chapter 7, we summarize the results and provide future directions.

Chapter 2

Research Journey

I have chosen C-STAR as my lab in my 3rd year at University and worked through multiple areas for my research.

Authentication Protocols: At first, I started with information security and authentication protocols. I worked on multi-factor authentication using public key cryptography.

Quantum Information Security: I always wanted to work in areas which have potential for industrial realization and starting up. Thinking that quantum information security has practical applications, I worked for two semesters on Quantum teleportation and quantum related information security protocols. But I liked working on classical information security more and so switched back.

Homomorphic Encryption: I worked for sometime on Homomorphic Encryption assisting my advisor and lab partner. That was my first experience of working on multiple solutions, breaking them, fixing the issues and trying new approaches. I am grateful for the time my professor spent with us on this problem as this has taught me an important thing about research, iterating on multiple solutions and quickly breaking each.

Secure Computation using uncertainties: I worked on implementing my professor's 1 in 4 bit protocol to achieve secure multi party computation. I tried generating bit flips in ethernet cable but could not get them reliably. Then I started working on protocols to achieve 1 in 4 bit flip using race conditions in Operating Systems. This is my final research topic and I built and implemented initial versions of such protocols while in college.

After being in the University campus for six and half years, I moved out for work but continued working on my research whenever I could find time. I registered every semester until now and tried to do work persistently albeit at a slow rate.

After some months, I realized experimental realizations of 1-4 bit protocol are not worth publishing as that would also need my professor's 1-4 bit protocol to be published but it's an elegant protocol and it would be great to patent that instead.

So I started working on implementing protocols in the existing literature using Operating Systems uncertainties. Claude Crepeau's original paper realizes secure multi party computation using both binary symmetric channels and erasure channels.I worked on implementing a binary symmetric channel by

leveraging randomness in operating systems and distributed systems. I managed to get some working protocols and experimental simulations, wrote a paper, and submitted it for publication at a conference. While one was a weak acceptance, the others were rejections. However I got constructive feedback that the binary symmetric channel simulated is not regulated and still has a random flip probability.

Since then, I worked on simulating erasure channels instead of binary symmetric channels.I worked on constructing multiple protocols, most of the initial versions not working and needing different directions. Finally, I arrived at a few working protocols with experimental implementations confirming them. I also implemented a generic C++ library which could take any uncertainty implementation and create secure computation.

Because this is an entirely new idea, most of the literature review, though helpful in providing a general understanding, did not give pointers on finding protocols. The amount of work needed on the implementation, operating systems and distributed systems was much more than the work on the information security side.

The following is the final product of my research work, the rest of this thesis goes into explaining it.

Chapter 3

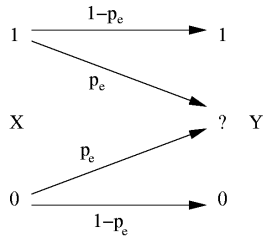
Architecture

This chapter goes through the architecture we constructed to get secure computation from uncertainties.

3.1 Step 1: Simulate an erasure channel

3.1.1 Erasure Channel

An erasure channel refers to a communication channel where, upon transmitting a set of data bits, there is a chance that some of those bits may be lost. Following diagram shows an erasure channel with an erasure probability of p(e)



3.1.2 Step 2: Get 1-2 oblivious transfer from erasure channel

We use Crepeau's passive/honest protocol from the literature [7].

1: Let b_0, b_1 be Sam's secret bits, let s be the bit Rachel wishes to see, and let k be the security parameter. Sam uniformly selects bits $c[1], \ldots, c[k]$, and transfers them through the oblivious transfer channel.

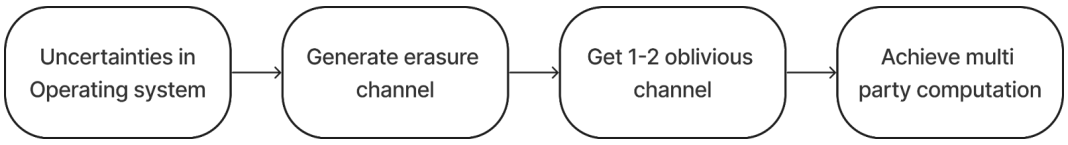
2: Rachel randomly picks i_0, i_1 such that she received $c[i_0]$, and didn't receive $c[i_1]$(for k large, i_0, i_1 will exist with high probability). She sends i_s, i_{1-s} to Sam.

3: Sam sends $b_0 \oplus c[i_s], b_1 \oplus c[i_{1-s}]$ to Rachel.

3.1.3 Step 3: Secure computation

For a given function, secure computation can be achieved from the already constructed 1-2 oblivious channel.

3.2 Pipeline



Chapter 4

Instruction Reordering Uncertainty

This chapter introduces the first protocol that implements the modeling approach outlined in Chapter 3. Specifically, we leverage CPU instruction reordering as a source of uncertainty to construct an erasure channel.

4.0.1 CPU instruction reordering

When developing multithreaded programs without synchronization, memory and instruction reordering are potential hazards to be aware of, though uncommon. To illustrate, we will examine a scenario involving reordering that is possible under the x86/64 architecture specification [11] [18]

Consider an example with two integer variables, X and Y, both initialized to 0. There are two separate processors that update X and Y to 1, respectively. After the updates, each processor stores the other variable into a new variable - the first processor stores Y in r1, and the second stores X in r2.

Processor 1	Processor 2
X = 1	Y = 1
r1 = Y	r2 = X

Intuitively, you would expect at least one of r1 or r2 to be set to 1, since the storing happens after X and Y are updated. However, this outcome is not guaranteed due to possible instruction reordering within each processor.

Specifically, a processor can reorder instructions as long as the end result is unchanged from its perspective. In this case, the first processor could delay the store of 1 to X until after loading Y into r1, since these are independent operations from its standpoint. Similarly, the second processor may load X into r2 before the store of 1 to Y takes effect.

If such reordering occurs simultaneously on both processors, it could result in r1 and r2 both retaining the initial value of 0, as illustrated below.

Processor 1	Processor 2
r1 = Y	r2 = X
X = 1	Y = 1

Although counterintuitive, this demonstrates how instruction reordering introduces uncertainty in concurrent programming. Building on this concept, the following protocol explores utilizing this uncertainty to simulate an erasure channel.

4.0.2 Protocol

This protocol utilizes the same construct as the previous example. There are two processors, operating on shared and private variables. Specifically, X and r1 are global integers initialized to -1. The first processor has a private integer m that is either 0 or 1. The second processor has a private integer r2 initialized to -2. The setup is illustrated by the following code:

```
// Global
X = -1
r1 = -1

// First Processor's private variables
m // either 0 or 1

// Second Processor's private variables
r2 = -2
```

The processors run the following code

Processor 1	Processor 2
X = m	while (r1 == -1){}
r1 = 1	r2 = X

Processor 1 contains two instructions - setting global variable X to the value of its private variable m, and setting the shared flag r1 to 1. Processor 2 waits for r1 to equal 1, then loads the value of X into its private variable r2.

Without instruction reordering, r2 would only receive the value of X after r1 is set to 1, which occurs after X is assigned the value of m. This should result in r2 always containing the value of m.

However, memory reordering can alter the actual instruction sequence, leading to unexpected results. The following tables show some possible orderings.

Processor 1	Processor 2
X = m	
r1 = 1	
	while (r1 == -1){}
	r2 = X

Table 4.1 Expected Order 1

Processor 1	Processor 2
	while (r1 == -1){}
X = m	
r1 = 1	
	r2 = X

Table 4.2 Expected Order 2

Processor 1	Processor 2
r1 = 1	
	while (r1 == -1){}
	r2 = X
X = m	

Table 4.3 Reorder 1

Processor 1	Processor 2
	r2 = X
X = m	
r1 = 1	
	while (r1 == -1){}

Table 4.4 Reorder 2

Expected orderings, for example Table 4.1 and Table 4.2 result in the expected output, with r2 containing m. Some reorderings (Table 4.3 and Table 4.4 for example) could specifically cause Processor 2 to load X into r2 before Processor 1 updates X to m. In this case, r2 would incorrectly retain the initial value of X, -1.

This protocol prevents the reliable transfer of m from Processor 1 to Processor 2 through global variable X.

4.0.3 Equivalence to erasure channel

This concurrent program can be characterized as an erasure channel, where messages from the sender are either received or lost by the receiver - who knows which outcome occurred.

Specifically, we can model Processor 1 as the sender and Processor 2 as the receiver. Processor 1 intends to communicate its private variable m as a message to Processor 2. Without reordering, Processor 2 reliably receives m through shared variable X. However, the uncertainty of instruction reordering simulates messages being ”erased” - Processor 2 loads the uninitialized value -1 instead when reordering alters the timing.

Since Processor 2 can detect if it received the valid message m or the marker value -1 indicating erasure, this satisfies the definition of an erasure channel. The non-determinism introduced through reordering essentially simulates message erasures and makes this an apt realization of the erasure channel abstraction.

In summary, the sender (Processor 1) attempts to transmit messages (values of m) that may either be received or lost by the receiver (Processor 2), modeling erasure channel characteristics.

4.1 Implementation

We implemented this protocol in C++ using two threads to represent the two processors. There are multiple possible implementation options - the processors could alternatively be modeled as processes with inter-process communication.

Our thread-based implementation adapts the instruction reordering example from [18], utilizing the same coordinated threading structure. However, while the cited source demonstrates memory reordering effects generically, our protocol leverages reordering specifically to simulate erasure channel characteristics.

The shared variables X and r1 are global variables, shown below:

```
// Globals
int X = -1;
int r1 = -1;
```

Shown below is the code executed by the sender thread:

```
void *sender(void *param)
{
    RandomGenerator random();
    while (true) {
        sem_wait(&startSender);

        int m = random.integer() %2; // message m

        X = m;

        r1 = 1;
        sem_post(&endSender);
    }
    return NULL;
}
```

The message m is randomly generated in the sender thread. Two semaphores coordinate the timing of the sender with the receiver thread.

The receiver code follows a similar structure:

```
void *receiver(void *param)
{
    RandomGenerator random();
    while (true) {
        sem_wait(&startReceiver);

        // wait until r1 is set to 1
        while (r1 != 1) { }

        r2 = X;
        sem_post(&endReceiver)
    }
    return NULL;
}
```

The main handler thread creates and coordinates the sender and receiver threads. It executes multiple iterations of the protocol by signaling the start semaphores and waiting on the end semaphores.

In this implementation, the handler is not an independent third party - its sole purpose is facilitating repeated protocol executions between the sender and receiver threads. This coordination could alternatively be managed by the sender and receiver threads directly.

4.1.1 Experimental Results

This protocol could lead to different number of erasures based on the processor, operating system, current system load and multiple other factors.

4.1.1.1 On macOS (OSX)

We executed this protocol on an Apple MacBook Pro with an ARM-based M2 processor and macOS Ventura version 13.6.2. Over one million iterations, this configuration yielded an average of one erasure per million protocol runs, as summarized in Table 4.5.

One variation is to introduce a random sleep delay at the beginning of each protocol iteration within both the sender and receiver threads. This helps desynchronize the threads' timing and increases the probability of instruction overlap and reordering. This resulted in one erasure per fifty thousand rounds, as seen in Table 4.6

Erasures	Number of rounds
1	1577632
2	1654436
3	3047709
4	3915530
5	4076525
6	6726489
7	7320902

Table 4.5 MacOS

Erasures	Number of rounds
1	13006
2	31545
3	40179
4	64473
5	88562
6	323782
7	342237

Table 4.6 MacOS Variation

4.2 Conclusion

In this chapter, we have presented an original technique leveraging memory or instruction reordering to construct an erasure channel model. The feasibility of this protocol is demonstrated by its successful implementation and ability to produce erasures under a real-world operating system.

Chapter 5

Race conditions and Scheduling Uncertainty

This chapter presents the second protocol adhering to the modeling framework detailed in Chapter 3. In this protocol, the nondeterminism of thread interleaving and processor scheduling is utilized to simulate an erasure channel.

5.0.1 Thread scheduling uncertainty

When concurrent threads or processes launch simultaneously, nondeterministic factors affect execution order and timing. For example, interrupts can pause a thread unpredictably, while competition for resources from other processes also introduces uncertainty. This indeterminacy regarding start order and interleaving duration cannot be resolved.

We leverage these sources of uncertainty to simulate erasures in an erasure channel protocol detailed below. As we will demonstrate, unpredictable interrupts and scheduling create ambiguity that can be modeled analogously to random message erasure events.

5.1 Protocol

This section presents two protocols that realize erasure channels leveraging nondeterminism in thread scheduling and execution.

5.1.1 Uncertain Thread Initialization Order

While conceptually simple, this clever protocol leverages uncertainty in the relative startup order of threads accessing a shared variable. Consider two threads with access to a global variable X. X is initialized to -1 at the start of each iteration. During every iteration, Thread 1 randomly updates X to its private variable m (either 0 or 1), while Thread 2 reads the value of X.

If Thread 1 executes before Thread 2, then Thread 2 will read the value set by Thread 1. However, if Thread 2 starts up first, the value it reads for X will be -1, the unitialized value.

Therefore, Thread 2 may either receive the message value from Thread 1, or read a known placeholder value indicating no valid message was present.

```
// Global
X = -1


// First thread
X = m // m is either 0 or 1


// Second thread
read X
```

5.1.1.1 Equivalence to erasure channel

This concurrent program can be characterized as an erasure channel, where messages from the sender are either received or lost by the receiver - who knows which outcome occurred.

Specifically, we can model Thread 1 as the sender and Thread 2 as the receiver. Thread 1 intends to communicate its private variable m as a message to Thread 2. Because thread execution order can change each time, thread 1 may read X after thread 2 writes or vice versia. And thread 2 thus receives m or -1 based on this.

Since Thread 2 can detect if it received the valid message m or the placeholder value -1 indicating erasure, this satisfies the definition of an erasure channel.

5.1.2 Thread Scheduling Uncertainty

This more sophisticated protocol utilizes nondeterminism in both thread execution order and runtime duration to realize an erasure channel.

Thread A aims to transmit N random binary messages to Thread B, where N is a known global integer. The threads share access to a global M-element array acting as a queue, with M ¡ N. Each queue slot stores a tuple (i, j) containing the message index i and value j.

The global queue array is initialized to (-1, -1) placeholder values. Thread A continually writes its N message tuples into the queue, wrapping around cyclically. Concurrently, Thread B tries to read messages from the global queue into its private M-element buffer, checking for non-placeholder values. This occurs atleast $K \geq N/M$ iterations, sufficient for Thread B to attempt reading N messages.

Due to ambiguous thread scheduling, Thread B may read the queue before, during, or after Thread A writes. This leads to uncertain number of messages being received - Of the N messages, Thread B ultimately receives a nondeterministic subset while losing others.

```
// Global

N // number of messages
M // size of queue
Queue[M][2] // queue of index, message

// First thread
m[N] // random binary messages

// copy message to nextIndex in the circular buffer Queue
for(i = 0 -> N)
    Queue[nextIndex] = (i, m[i])

// Second thread
for(j = 0 -> K)
    for (i = 0 -> M)
        store Queue[i][1] message with Queue[i][0] index
```

The flow described above provides high-level pseudocode illustrating the key mechanisms of this protocol. Later in the Implementation section, we will examine how the parameters K, M, and N impact protocol behavior. Additionally, potential enhancements like introducing intentional staggering between the threads are explored to improve uncertainty.

5.1.2.1 Equivalence to erasure channel

Similar to the previous protocol, we define Thread A as the sender and Thread B as the receiver. The sender attempts to transmit N binary messages to the receiver. Due to ambiguous thread timing, the receiver only actually reads some of these N messages. From the sender side, it is ambiguous which messages were lost by the receiver. However, based on index gaps from 0 to N, the receiver knows definitively which messages were erased in transmission. Therefore, the thread uncertainty results in erasures seen by the receiver but not identifiable by the sender, thus modeling an erasure channel.

OS	Method	Erasure percent for 10k rounds
Ubuntu 20.04	Thread 1 first	1%
Ubuntu 20.04	Thread 2 first	99.9%
macOS Ventura	Thread 2 first	60%
macOS Ventura	Thread 1 first	11%
Windows 11	Thread 1 first	0.03%
Windows 11	Thread 2 first	96%

Table 5.1 Comparision on various Operating Systems 1

5.2 Implementation

5.2.1 Uncertain Thread Initialization Order

The main handler thread creates and coordinates the sender and receiver threads. It executes multiple iterations of the protocol by signaling the start semaphores and waiting on the end semaphores.

In this implementation, the handler is not an independent third party - its sole purpose is facilitating repeated protocol executions between the sender and receiver threads. This coordination could alternatively be managed by the sender and receiver threads directly. Alternatively, the two parties could be implemented as processes too, as threads are essentially processes at the system level.

Table 5.1 shows the following results in a tabular format.

5.2.1.1 Ubuntu 20.04

With Thread A starting first, over 100,000 iterations we observed an average of 1,100 erasures, yielding a 1% erasure rate. Reversing the order to launch Thread B first resulted in an expected 99.9% erasure rate, with around 98,931 erasures per 10,000 messages sent.

5.2.1.2 macOS Ventura

On macOS Ventura 13.6.2, erasure frequency differed significantly from Ubuntu. Across 100,000 rounds, up to 60,461 erasures occurred, representing a 60% erasure rate. With Thread 2 starting first, 100,000 rounds yielded approximately 11,000 erasures, reducing the rate to 11%.

5.2.1.3 Windows 11

Under Windows 11, only 0.03% of messages were erased over 100,000 rounds, with a maximum of 3,406 erasures. As expected, prioritizing Thread 2 increased the erasure rate drastically to 96%, with 96,639 erasures per 100,000 rounds.

OS	Erasure percent for 10k rounds
Ubuntu 20.04	50.5%
macOS Ventura	44%
Windows 11	54.3%

Table 5.2 Comparision on various Operating Systems 2

5.2.2 Thread Scheduling Uncertainty

By adjusting key parameters N, K, and the sender's stagger delay when the queue is full, we can modulate the erasure rate from under 1% up to over 99%. The following experimental results set N=100,000 and K=500,000, with a fixed sender stagger of 300 instructions upon queue being full. This configuration yielded a consistent erasure rate between 45-55% across all the evaluated architectures.

Table 5.2 shows the following results in a tabular format.

5.2.2.1 Ubuntu 20.04

Over 100,000 iterations we observed an average of 50552 erasures, yielding a 50.5% erasure rate.

5.2.2.2 macOS Ventura

On macOS Ventura 13.6.2, the results were similar. Across 100,000 rounds, up to 43644 erasures occurred, representing a 44% erasure rate.

5.2.2.3 Windows 11

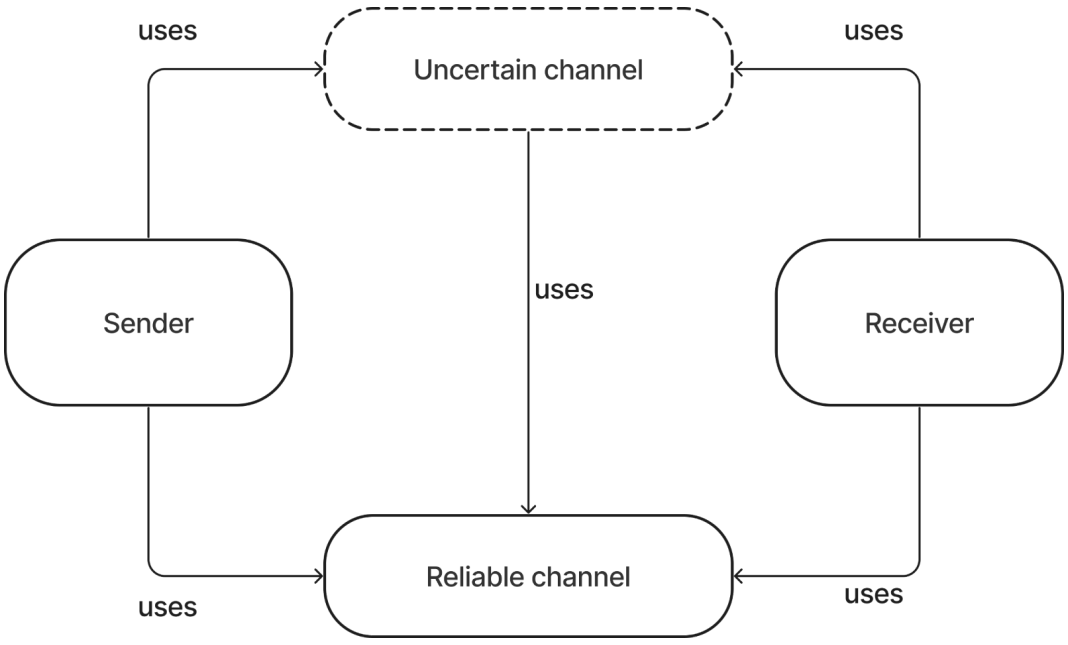
Under Windows 11, 54.3% of messages were erased over 100,000 rounds, with 54,398 erasures.

Chapter 6

Library

We have implemented a generic C++ library which can be used to plug in any uncertain channel (and any reliable channel). Following is a brief architecture diagram for the same. I have plugged in the previously described protocols into this library to achieve secure computation

6.1 Library



6.1.1 Components

Following are examples of how the components in the library look like.

6.1.1.1 Receiver

The receiver (and the sender) have access to a reliable and an uncertain channel.

```
class Receiver
{
    private:
        bool d_index;
        ReliableChannel* d_reliableChannel;
        UncertainChannel* d_uncertainChannel;

    public:
        Receiver(bool i, ReliableChannel* reliableChannel,
                 UncertainChannel* uncertainChannel);
        void startProtocol();
};
```

6.1.1.2 UncertainChannel

The uncertain channel has access to a reliable channel. It supports generation, sending and receiving bits. It also supports retry mechanism for cases when there are no erasures or vice versia.

```
class UncertainChannel
{
    public:
        UncertainChannel(int numInitialBits,
                         ReliableChannel* reliableChannel);
        virtual void generateBits();
            // optional, by default generates
            // random numInitialBits number of bits
        virtual void sendBits() = 0;
            // by sender only
        virtual void receiveBits() = 0;
            // by receiver only
```

```
        virtual std::vector<int> getFinalBits();
            // optional, by default returns d_initialBits
        virtual std::vector<int> getInitialBits();
            // optional, by default returns d_initialBits
        virtual int getNumTries();
            // for retries
    protected:
        int d_numInitialBits;
        std::vector<int> d_initialBits;
        ReliableChannel* d_reliableChannel;
        int d_numTries;
};
```

6.1.2 Example Run

Following is an example run of protocol 2 channel (uncertain channel) with tcp sockets (reliable channel) to implement 1-2 Oblivious transfer.

The sender runs the program with two messages

```
bhanudev@Bhanu-PC:~/compsec/thesis/2-1-from-uncertainties$ ./race-conditions-ot  rc alice 0 1
1 1 0 0 1 0 1 1 1 0  Num tries 3
From bob received 4 7
Sending 1 0
Protocol run successfully
```

The receiver runs the program asking for a single messsage

```
bhanudev@Bhanu-PC:~/compsec/thesis/2-1-from-uncertainties$ ./race-conditions-ot rc bob 1
-1 -1 -1 -1 -1 0 1 1 -1 -1  Num tries 3
Data: 7 Erasure: 4
Received from alice 1 0
Received value is 1
```

Chapter 7

Conclusions

The contributions in this thesis are as follows:

- A novel idea proposing that operating systems uncertainties can be used to achieve fundamental cryptographic protocols like Secure computation and Oblivious Transfer
- Protocols which use this idea, implemented on various operating systems proving the feasibility of these.
- A library implementing this pipeline with the uncertain channel provided as a pluggable component.
- These protocols provide information theoretic security whereas the current public key based oblivious transfer protocols provide computational security based on hard to solve problems like factorization

Future directions would be to focus on adapting these protocols for malicious adversaries and investigating other uncertainties like incorrect cache coherence and distributed systems uncertainties.

Related Publications

- Bhanu Dev Chaluvadi and Kannan Srinathan. Dancing with Uncertainties: A meritorious case for race conditions and inconsistencies in operating systems [Under Preparation]

Bibliography

[1] K. Antoniadis, P. Blanchard, R. Guerraoui, and J. Stainer. The entropy of a distributed computation random number generation from memory interleaving. *Distrib. Comput.*, 31(5):389–417, Oct. 2018.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, 2010. USENIX Association.

[3] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Producing scheduling that causes concurrent programs to fail. In *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '06, page 37–40, New York, NY, USA, 2006. Association for Computing Machinery.

[4] L. Chen. Oblivious transfer from noisy channel.

[5] A. Colesa, R. Tudoran, and S. Banescu. Software random number generation based on race conditions. In *2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 439–444, 2008.

[6] C. Crépeau. Equivalence between two flavours of oblivious transfers. In C. Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 350–354, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[7] C. Crepeau and J. Kilian. Achieving oblivious transfer using weakened security assumptions. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pages 42–52, 1988.

[8] C. Crépeau, K. Morozov, and S. Wolf. Efficient unconditional oblivious transfer from almost any noisy channel. In C. Blundo and S. Cimato, editors, *Security in Communication Networks*, pages 47–59, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[9] D. Evans, V. Kolesnikov, and M. Rosulek. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2(2-3):70–246, 2018.

[10] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 215–228, New York, NY, USA, 2011. Association for Computing Machinery.

[11] Intel. Intel x86/64 architecture specification. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html/, 2012.

[12] S. Khoshnood, M. Kusano, and C. Wang. Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 165–176, New York, NY, USA, 2015. Association for Computing Machinery.

[13] J. Kilian. Founding crytpography on oblivious transfer. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 20–31, New York, NY, USA, 1988. Association for Computing Machinery.

[14] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGOPS Oper. Syst. Rev.*, 42(2):329–339, Mar. 2008.

[15] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 329–339, New York, NY, USA, 2008. Association for Computing Machinery.

[16] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. pages 267–280, 01 2008.

[17] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, 1983.

[18] J. Preshing. Memory reordering caught in the act. https://preshing.com/20120515/memory-reordering-caught-in-the-act/, 2012.

[19] C. Pu and A. Leff. Replica control in distributed systems: as asynchronous approach. In *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 377–386, 1991.

[20] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. Predicting and preventing inconsistencies in deployed distributed systems. *ACM Trans. Comput. Syst.*, 28(1), Aug. 2010.

[21] A. C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982.