Communication Detection and Data Transfer Event Synchronization from Dual Trace

by

Huihui Nora Huang
B.Sc., Nanjing University of Aeronautics and Astronautic, 2003
M.Sc., Nanjing University of Aeronautics and Astronautic, 2006

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Communication Detection and Data Transfer Event Synchronization from Dual Trace

by

Huihui Nora Huang
B.Sc., Nanjing University of Aeronautics and Astronautic, 2003
M.Sc., Nanjing University of Aeronautics and Astronautic, 2006

Supervisory Committee

Dr. German. Supervisor Main, Supervisor
(Department of Same As Candidate)

Dr. M. Member One, Departmental Member
(Department of Same As Candidate)

Dr. Member Two, Departmental Member
(Department of Same As Candidate)

Dr. Outside Member, Outside Member
(Department of Not Same As Candidate)

Supervisory Committee

Dr. German. Supervisor Main, Supervisor
(Department of Same As Candidate)

Dr. M. Member One, Departmental Member
(Department of Same As Candidate)

Dr. Member Two, Departmental Member
(Department of Same As Candidate)

Dr. Outside Member, Outside Member
(Department of Not Same As Candidate)

ABSTRACT

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vii**

List of Figures **viii**

Acknowledgements **ix**

Dedication **x**

1 Introduction **1**

1.1 Define the Problem 2

1.2 Obtain Background Knowledge 2

1.3 Model the Communication Channels 2

1.4 Develop the Dual Trace Synchronization Algorithms 3

1.5 Apply the Channel Models to Windows APIs 3

1.6 Implement the Trace Synchronization Algorithms 3

1.7 Evaluate the Communication Channel Rebuilt Feature in Atlantis 3

2 Background **4**

2.1 Software Security 4

2.2 Software Vulnerability Detection 5

2.3 Assembly Level Trace 5

2.4 Software Interaction 5

2.5 Atlantis 5

2.6 Windows Communication Foundation 6

3 Modeling 7

3.1 Communication of Two Programs 7

3.1.1 Terminology 8

3.1.2 Definition 8

3.2 Communication Categorization and Communication Methods 10

3.2.1 Reliable Communication 10

3.2.2 Unreliable Communication 10

3.2.3 Communication Methods 11

3.3 Communication Identification Strategy 14

3.4 Communication Methods' Implementation in Windows 15

3.4.1 Windows Calling Convention 16

3.4.2 Named Pipes 16

3.4.3 Message Queue 19

3.4.4 TCP and UDP 20

3.5 Communication Event Locating in Assembly Execution Traces 22

3.5.1 Assembly Execution Trace 22

3.5.2 Event Information Retrieval 23

4 Communication Identification Algorithms and Prototype Implementation 24

4.1 Algorithms for Communication Identification in Dual-trace 24

4.1.1 Event Locating Algorithm 24

4.1.2 Endpoint and Streams Identification Algorithm 25

4.1.3 Channel Identification Algorithm 27

4.1.4 Transported Stream Data Verification 27

4.1.5 Transported Event Data Verification 27

4.1.6 Data Structure for Identified Communications 27

4.2 Channel Information Retrieval Algorithm for UDP 32

4.3 Channel Information Retrieval Algorithm for Named pipe 35

4.4 Communication Identification Feature Prototype 38

5 Feature Prototype On Atlantis 39

5.0.1 User Defined Channel Type By Json 39

5.0.2 Communication Event Searching 43

5.0.3 Navigation From Channel Search Result to Dual-Trace 44

6 Evaluation **46**

6.0.1 Experiment Verification Design 46

6.0.2 Result 46

7 Conclusions **49**

7.1 Limitations 49

7.1.1 Event Status: Success or Fail 49

7.1.2 Match Events Distinguishing 49

7.1.3 Match Events Ordering 50

7.1.4 Buffer Sizes Of Sender and Receiver Mismatch 50

A Additional Information **51**

Bibliography **52**

List of Tables

Table 3.1 Communication Methods Discussed in This Work 10
Table 3.2 Function List for Synchronous Named Pipe 17
Table 3.3 Function List for Asynchronous Named Pipe 18
Table 3.4 Function List for Synchronous MSMQ 19
Table 3.5 Function List for Asynchronous MSMQ with Callback 19
Table 3.6 Function List for Asynchronous MSMQ without Callback 20
Table 3.7 Function APIs for TCP and UDP 21

List of Figures

Figure 3.1 General Communication Model 9

Figure 3.2 Data Transfer Scenarios for Named Pipe 11

Figure 3.3 Data Transfer Scenarios for Message Queue 12

Figure 3.4 Data Transfer Scenarios for TCP 13

Figure 3.5 Data Transfer Scenarios for UDP 14

Figure 3.6 Channel Open Process for a Named Pipe 18

Figure 3.7 Channel Open Process for a Message Queue 20

Figure 3.8 Channel Open Model for TCP and UDP 22

Figure 5.1 Add function to a Communication type from Functions View 40

Figure 5.2 Dialog to input information for a function adding to a communication type . 41

Figure 5.3 New View: Communication Type View 43

Figure 5.4 Right Click menu to navigate to send and receive event in the traces 43

Figure 6.1 Defined clientsend and serversend communication types in Communication View 47

Figure 6.2 the search result of clientsend communication type 47

Figure 6.3 the search result of the serversend communication type 47

Figure 6.4 instruction view and memory view updated correctly 48

ACKNOWLEDGEMENTS

I would like to thank:

my cat, Star Trek, and the weather, for supporting me in the low moments.

Supervisor Main, for mentoring, support, encouragement, and patience.

Grant Organization Name, for funding me with a Scholarship.

> *I believe I know the only cure, which is to make one's centre of life inside of one's self, not selfishly or excludingly, but with a kind of unassailable serenity-to decorate one's inner house so richly that one is content there, glad to welcome any one who wants to come and stay, but happy all the same in the hours when one is inevitably alone.*
>
> Edith Wharton

DEDICATION

Just hoping this is useful!

Chapter 1

Introduction

Many network application vulnerabilities occur not just in one application, but in how they interact with other systems. These kinds of vulnerabilities can be difficult to analyze. Dual-trace analysis is one approach that helps the security engineers to detect the vulnerabilities in the interactive software. A dual-trace consist of two execution traces that are generated from two interacting applications. Each of these traces contains information including CPU instructions, register and memory changes of the running application. Communication information of the interacting applications is captured as the register or memory changes on their respective traced sides.

This work is focusing on helping reverse engineers for interacting software vulnerabilities detection. We first investigated and modeled four types of commonly used channels in Windows communication foundation in order to help the reverse engineers to understand the APIs, the scenarios and the assembly trace related perspectives of these channels. Then we built a tool prototype for the communication event locating and visualization of dual-traces. Finally, we design an experiment to test our prototype and evaluate its practicality.

add an section to summarize the conclusion later

The Methodology used for this work composed of 7 major steps. To make this work executable, 1)I defined the problem by understanding the requirement from our research partner DRDC. 2) I obtained the related background knowledge by literature review. Then 3) I model the abstract communication channels. Based on these channel models,4) I develop algorithms to synchronize the communication events happen in the channel. After that, 5) I match the real channels used in Windows Communication Foundation to my channel models, verify their consistency with my models. Finally 6)I implement the synchronization algorithms for the dual-trace analysis and verify them by the dual-traces from DRDC.

1.1 Define the Problem

A dual-trace consists of two execution traces that are generated from two interacting applications. The trace analysis is based only on the assembly level execution trace which contain the instructions and memory change of a running application. Beside all the factors in single trace analysis, dual-trace analysis has to analyze the communications of the applications in the traces. A communication between two applications including the communication channel open, all data exchanging events, the communication channel close. Correspondingly, a full communication definition in the dual-trace should consist of the channel opening events in both sides, data sending and receiving events, and the the channel closing events in both sides. Each of these events consist of function call and related data from the memory record. In some cases there might be some events lacking from the trace, such as no data exchange after a channel is open, or the traces end before the channel was closed. However, the channel open is critical, without that there is no way to locate all other events in the traces. The goal communication analysis of dual-trace is to rebuild all the user concerned communication channels from the dual-trace.

1.2 Obtain Background Knowledge

I did a some background reading in the reverse engineering filed, focusing more on the vulnerabilities detection domain to better understand the current state and needs. In addition, to locate the communication event of the dual-trace, I need to investigate the communication methods' APIs to understand their structure in the assembly level traces. I need to know how the functions for channel setup and the functions for messages sending/receiving work. The system functions I was looking for is in C++ level. I have to know the C++ function names, related parameters, return value and so on. Furthermore, to understand their structure in the assembly level trace, I have to know the calling conventions in assembly, such registers/memory for parameters or return value.

1.3 Model the Communication Channels

There are two abstract models for communication based on the communication behavior. One is the order guaranteed communication model and the other is order in-guaranteed communication model. I define how the communication happens as well as all the data send/receive scenario in each model. Later on the real communication channels will be categorized into these two models.

1.4 Develop the Dual Trace Synchronization Algorithms

1.5 Apply the Channel Models to Windows APIs

I investigate 4 types of communication channels in Windows Communication Foundation and match each of them to the developed channel model. These 4 types are Only Named pipe, MQMS, HTTP, and TCP/UDP socket. The matching includes two steps: 1. Put each type of communication channel in the modeling categories by verify the existence of the message send/receive scenario. 2. Define the function callings for each event types in the channel, such as channel opening and closing, data sending and receiving.

1.6 Implement the Trace Synchronization Algorithms

1.7 Evaluate the Communication Channel Rebuilt Feature in Atlantis

Chapter 2

Background

This section introduces several background knowledge or information that related to this work. First I describe what is software security and how important it is as well as our previous approach to assist detection of software vulnerabilities by assembly level trace analysis. Second, I introduce the general assembly level trace as well as some tracer to generate it. Third, I discuss how software interaction affect the behavior of the software and how they related to the software vulnerabilities. Then I talk about Windows Communication Foundation in which the communication channels type used are targeted by this work. Finally, we mention some important Windows function calling conventions without which you can not picture what the function calls look like in the assembly level.

2.1 Software Security

The internet grows incredibly fast in the past few year. More and more computers are connected to it in order to get service or provide service. The internet as a powerful platform for people to share resource, meanwhile, introduces the risk to computers in the way that it enable the exploit of the vulnerabilities of the software running on it. Accordingly, the emphasize placed on computer security particularly in the field of software vulnerabilities detection increases dramatically. It's important for software developers to build secure applications. Unfortunetely, this is usually very expensive and time consuming and somehow impossible. On the other hand, finding issues in the built applications is more important and practical. Howerver this is a complex process and require deep technical understanding in the perspetive of reverse engineering.[2].

2.2 Software Vulnerability Detection

A common approach to detect existing vulnerabilities is fuzzing testing, which record the execution trace while supplying the program with input data up to the crash and perform the analysis of the trace to find the root cause of the crash and decide if that is a vulnerability[1]. Execution trace can be captured in different levels, for example object level and function level. But my research only focus on those that captured in instruction and memory reference level. There are two main reasons for analysis system-level traces. First, it is for analysis of the software provided by vendor whose source code are not available. The second one is that low level trace are more accurately reflect the instructions that are executed by multicore hardware[9].

2.3 Assembly Level Trace

There are many tools that can trace a running program in assembly instruction level. IDA pro [3] is a widely used tool in reverse engineering which can capture and analysis system level execution trace. Giving open plugin APIs, IDA pro allows plugin such as Codemap [6] to provide more sufficient features for ”run-trace” visualization. PIN[5] as a tool for instrumentation of programs, provides a rich API which allows users to implement their own tool for instruction trace and memory reference trace. Other tools like Dynamic **??** and

2.4 Software Interaction

Applications nowaday do not alway work isolately, many software appear as reticula collaborating systems connecting different modules in the network[**?**] which make the discovery of vulnerabilities even harder. The communication and interaction between modules affect the behaviour of the software. Without regarding to the synergy information , analysis of the isolated execution trace on a single computer is usually futile.

2.5 Atlantis

Applications nowaday do not alway work isolately, many software appear as reticula collaborating systems connecting different modules in the network[**?**] which make the discovery of vulnerabilities even harder. The communication and interaction between modules affect the behaviour of the software. Without regarding to the synergy information , analysis of the isolated execution trace on a single computer is usually futile.

2.6 Windows Communication Foundation

We limited our research only on the communication types used in Windows Communication Foundation(WCF) for now. Windows Communication Foundation (WCF) is a framework for building service-oriented applications. Using WCF, developers can send data as asynchronous or asynchronous messages from one service endpoint to another. We are not going deep into the details of this framework but only mention the most common communication methods it supports in its messaging layer. The messaging layer in WCF is composed of channels. A channel is a component that processes a message in some protocol. There are two types of channels in WCF: transport channels and protocol channels. In this work we only care about transport channels. Transport channels read and write messages from the network. Examples of transports are named pipes, MSMQ, TCP/UDP or HTTP, all of which are involved in the scope of this work.

Chapter 3

Modeling

The goal of this modelling section is to 1) define the communication between two running programs. 2) design the communication identification strategy. 3) discuss the implementation of the communication methods in Windows 4) understand the dual-trace of two programs in the perspective of communication and how to retrieve the event information of the communications from the dual-trace

With this model, it is able to decide how communications being identified from the dual-trace and how to present them to the user.

3.1 Communication of Two Programs

This section describe my understanding of communications happen between two running programs in the real world. The communication in this work is data transfer activities between two running programs through a specific channel. Some collaborative activities between the programs such as remote procedure call is out of the scope of this research. Communication among multiple programs (more than two) is not discussed in this work. The channel can be reopened again to start new communications after being closed. However, the reopened channel will be treated as a new communication. The way that I define the communication is leading to the communication identification in the dual-trace. So the definition is not about how the communication works but what it looks like. The definition of the communication consist of terminology and definition.

3.1.1 Terminology

Endpoint:
An instance in a program at which a stream of data are sent or received (or both). It usually is identified by the handle of a specific communication method in the program. Such as a socket handle of TCP or UDP or a file handle of the named piped channel.

Channel:
A conduit connected two endpoints through which data can be sent and received

Channel open event:
Operation to create and connect an endpoint to a specific channel

Channel close event:
Operation to disconnect and delete the endpoint from the channel.

Send event:
Operation to send a trunk of data from one endpoint to the other through the channel.

Receive event:
Operation to receive a trunk of data at one endpoint from the other through the channel.

Channel open stream:
A set of all channel open events regarding to a specific endpoint.

Channel close stream:
A set of all channel close events regarding to a specific endpoint.

Send stream:
A set of all send events regarding to a specific endpoint.

Receive stream:
A set of all receive events regarding to a specific endpoint.

Stream:
A stream consist of a channel open stream, a channel close stream, a send stream and a receive stream. All of these streams regard to the same endpoint.

3.1.2 Definition

With the defined terminologies, a communication consist of two endpoints to a channel, each endpoint is corresponding to a stream. A stream contains one or more channel open events, one or zero send stream, one or zero receive stream and one or more channel close events. A visualized definition can be found in Figure3.1. The event numbers in this figure do not representing the exact number of a real communication but only an example. There are many communication methods for the data transfer communications, but all of them are compatible to this communication definition.

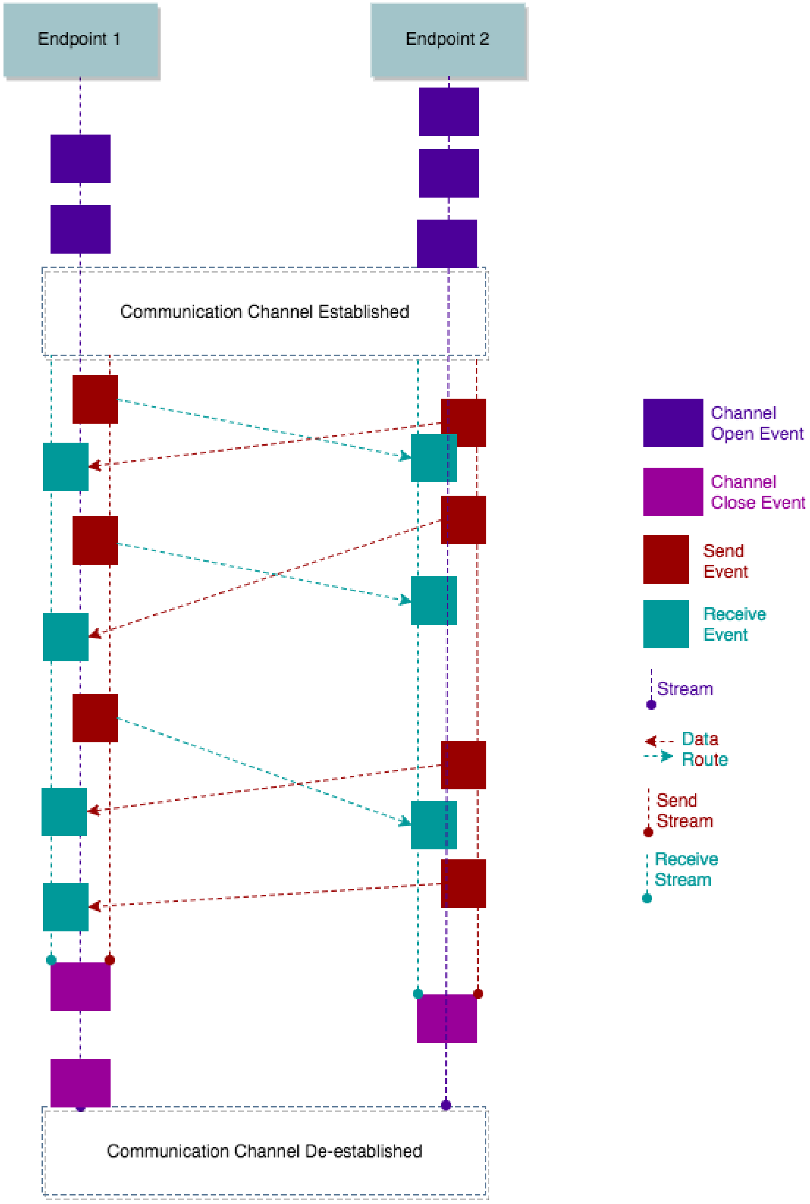


Figure 3.1: General Communication Model

3.2 Communication Categorization and Communication Methods

This model divides all communication methods in two groups: reliable and unreliable communications. The reason to divide the communication methods into these two categories is that the send and receive streams or send and receive events matching strategies are respective for these two categories. In the first following two subsections, I summarize the characteristics of these two communication categories. The communication methods list in Table3.1 will be discussed further to provide more concrete comprehension. Not all communication methods in the real world are being leveraged in this thesis. However, this model is supposed to be applicable to other communication methods.

Table 3.1: Communication Methods Discussed in This Work

Reliable Communication	**Unreliable Communication**
Named Pipes	Message Queue
TCP	UDP

3.2.1 Reliable Communication

A reliable communication guarantees the data being sent by one endpoint of the channel always received lossless and in order to the other endpoint. With this property, the send data union in the send stream of one endpoint should equal to the receive data union in the receive stream of the other endpoint. The send data union is the conjunction of the data trunks in all send events in the send stream by the event time ordering. The receive data union is the conjunction of the data trunks in all receive events in the receive stream by the event time ordering. Therefore, the send and receive data verification should be in send and receive stream level by comparing the send data union of one endpoint to the receive data union of another.

3.2.2 Unreliable Communication

An unreliable communication does not guarantee the data being send always arrive the receiver. Moreover, the data packets can arrive to the receiver in any order. However, the bright side of unreliable communication is that the packets being sent are always arrived as the origin packet, no data re-segmentation would happen. Accordingly, the send and receive data verification should be done by matching the data packets in a send event to a receive event on the other side.

3.2.3 Communication Methods

In this section, I describe the mechanism and the basic data transfer characteristics of each communication method in Table3.1 briefly. Moreover, data transfer scenarios are represented correspondingly in diagrams for each communication method.

Named Pipe

In computing, a named pipe provides FIFO communication mechanism for inter-process communication. It allows two programs send and receive message through the named pipe.

The basic data transfer characteristics of Named Pipe are:

- Bytes received in order
- Bytes sent as a whole trunk can be received in segments
- No data duplication
- Only the last trunk can be lost

Based on these characteristics, the data transfer scenarios of Named pipe can be summarized in Figure3.2.

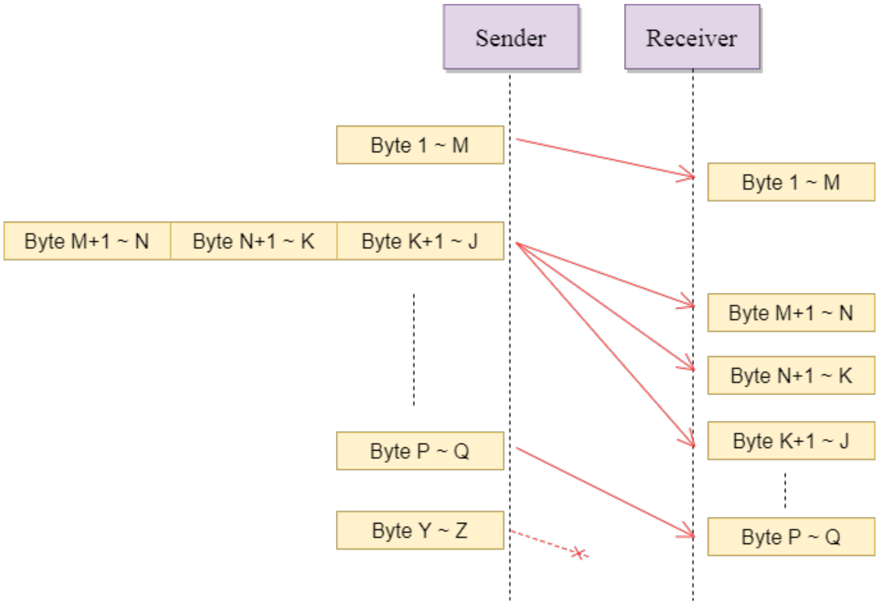


Figure 3.2: Data Transfer Scenarios for Named Pipe

Message Queue

Message Queuing (MSMQ) is a communication method to allow applications which are running at different times across heterogeneous networks and systems that may be temporarily offline can still communicate with each other. Messages are sent to and read from queues by applications. Multiple sending applications can send messages to and multiple receiving applications can read messages from one queue.[8] The applications are the endpoints of the communication. In this work, only one sending application versus one receiving application case is considered. Multiple senders to multiple receivers scenario can always be divided into multiple sender and receiver situation. Both endpoints of a communication can send to and receive from the channel.

The basic data transfer characteristics of Message Queue are:

- Bytes sent in packet and received in packet, no bytes re-segmented
- Packets can lost
- Packets received in order
- No data duplication

Based on these characteristics, the data transfer scenarios of Message Queue can be summarized in Figure3.3.

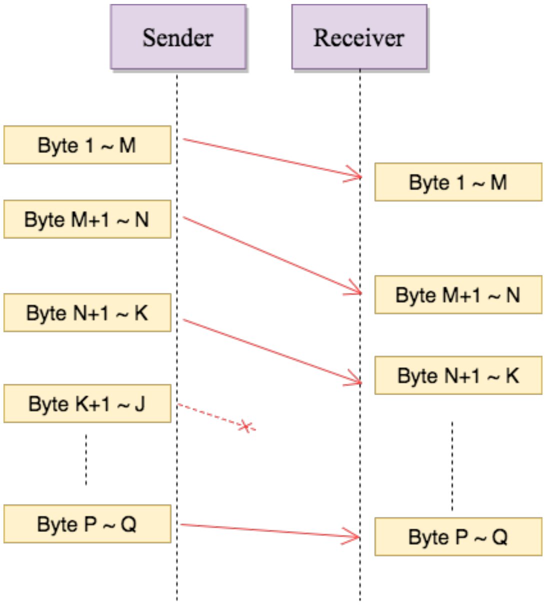


Figure 3.3: Data Transfer Scenarios for Message Queue

TCP

TCP is the most fundamental reliable transport method in computer networking. TCP provides reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts in an IP network. The TCP header contains the sequence number of the sending octets and the acknowledge sequence this endpoint is expecting from the other endpoint(if ACK is set). The retransmission mechanism is based on the ACK.

The basic data transfer characteristics of TCP are:

- Bytes received in order
- No data lost(lost data will be retransmitted)
- No data duplication
- Sender window size is different from receiver's window size, so packets can be re-segmented

Based on these characteristics, the data transfer scenarios of TCP can be summarized in Figure3.4.

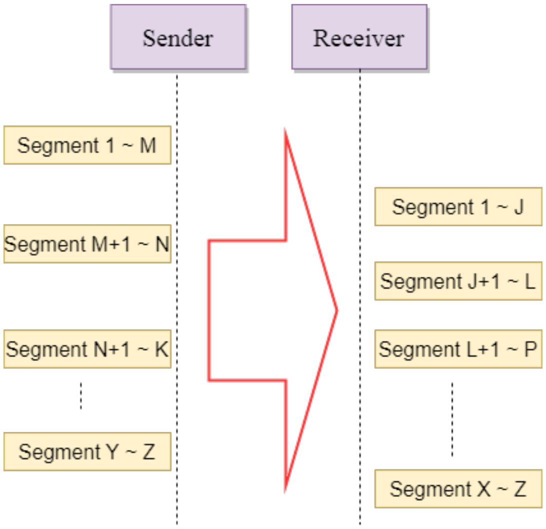


Figure 3.4: Data Transfer Scenarios for TCP

UDP

UDP is a widely used unreliable transmission method in computer networking. It is a simple protocol mechanism, which has no guarantee of delivery, ordering, or duplicate protection. This transmission method is suitable for many real time systems.

The basic data transfer characteristics of UDP are:

- Bytes sent in packet and received in packet, no re-segmentation
- Packets can lost
- Packets can be duplicated
- Packets can arrive receiver out of order

Based on these characteristics, the data transfer scenarios of UDP can be summarized in Figure3.5.

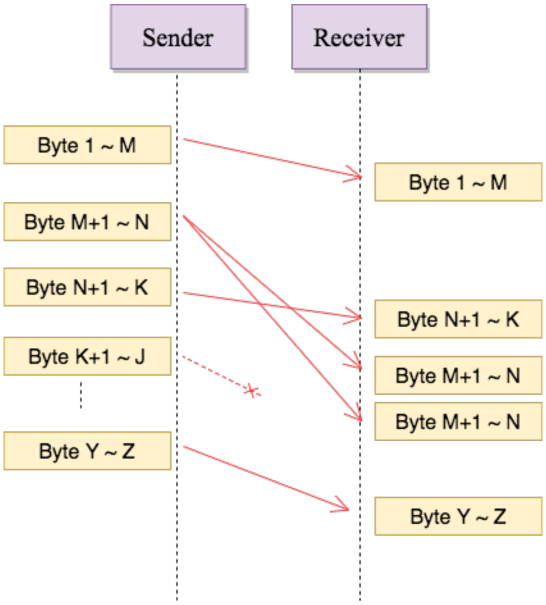


Figure 3.5: Data Transfer Scenarios for UDP

3.3 Communication Identification Strategy

In the first section of this chapter, I define what is a communication. The identification of the communications from a dual-trace should be able to identify the concerned communications as well as all the components defined in it. The section section in this chapter investigate the characteristics of the communication methods which is the key factor affecting the verification of the transferred data of a communication. From the trace analysis point of view, channel open events are essential to identify an endpoint while all other events are optional in the identification.

The identification contains seven major steps:

- Locate all events

- Identify the endpoints from the channel open events
- Group the related events into streams, send streams and receive streams, then bind them to the corresponding endpoint
- Match the endpoints to the communications
- For reliable communication, verify if the send data union in the stream of one endpoint match the receive data union in the stream of the other endpoint; For unreliable communication, try to match the send and receive events of both endpoints
- Construct and organize all the retrieved data of the communication
- Present the identified communications to the user

3.4 Communication Methods' Implementation in Windows

In this section, the implementations of the four communication methods in Windows system are investigated. The goal of this investigation is to determine the system functions for the events in the communication definition and summarize the necessary parameters of all the communication events to further identify a communication. Each function call will be considered as an event. The channel opening mechanism is essential for identifying the endpoints and further the communication. Hence the channel opening mechanisms of each method are described in detail and represented in diagrams.

I reviewed the Windows APIs of the communication methods and their example code. For each communication method, a system function list is provided for reference. These lists contain function names, essential parameters. These functions are supported in most Windows operating systems, such as Windows 8, Window 7.

Windows API set is very sophisticated and multiple solutions are provided to fulfil a communication method. It is impossible to enumerate all solutions for each communication method. I only give the most basic usage provided in Windows documentation. Therefore, the provided system function lists for the events should not be considered as the only combination or solution for each communication method. With the understanding of the model, it should be fairly easy to draw out lists for other solutions or other communication methods.

Moreover, the instances of this model only demonstrate Windows C++ APIs. This model may be generalizable to other operating systems with the effort of understanding the APIs of those operating systems.

3.4.1 Windows Calling Convention

The Windows calling convention is important to know in this research. The communication identification relies not only on the system function names but also the key parameter values. In the assembly level execution traces, the parameter values is captured in the memory changes of the instructions. The memory changes are recognized by the register names or the memory address. The calling convention helps us to understand where the parameters are stored so that we can find them in the memory change map in the trace.

Calling Convention is different for operating systems and the programming language. Based on the need of this work, we only list the Microsoft* x64 calling convention for interfacing with C/C++ style functions:

1. RCX, RDX, R8, R9 are used for integer and pointer arguments in that order left to right.

2. XMM0, 1, 2, and 3 are used for floating point arguments.

3. Additional arguments are pushed on the stack left to right. . . .

4. Parameters less than 64 bits long are not zero extended; the high bits contain garbage.

5. Integer return values (similar to x86) are returned in RAX if 64 bits or less.

6. Floating point return values are returned in XMM0.

7. Larger return values (structs) have space allocated on the stack by the caller, and RCX then contains a pointer to the return space when the callee is called. Register usage for integer parameters is then pushed one to the right. RAX returns this address to the caller.

3.4.2 Named Pipes

In Windows, a named pipe is a communication method for the pipe server and one or more pipe clients. The pipe has a name, can be one-way or duplex. Both the server and clients can read or write into the pipe.[7] In this work, I only consider one server versus one client communication. One server to multiple clients scenario can always be divided into multiple server and client communications thanks to the characteristic that each client and server communication has a separate conduit. The server and client are endpoints in the communication. We call the server “server endpoint” while the client “client endpoint”. The server endpoint and client endpoint of a named pipe share the same pipe name, but each endpoint has its own buffers and handles.

There are two modes for data transfer in the named pipe communication method, synchronous and asynchronous. Modes affect the functions used to complete the send and receive operation. I list the related functions for both synchronous mode and asynchronous mode. The create channel functions for both modes are the same but with different input parameter value. The functions for send and receive message are also the same for both cases. However, the operation of the send and receive functions are different for different modes. In addition, an extra function *GetOverlappedResult* is being called to check if the sending or receiving operation finish, the output message will be stored in the overlap structure whose memory address saved in the function's output parameter Overlap Structure Address. Table3.2 lists the functions for synchronous mode while Table3.3 lists the functions for the asynchronous mode for a Named pipe communication.

Table 3.2: Function List for Synchronous Named Pipe

Event	**Server Endpoint**		**Client Endpoint**	
	Function	**Parameters**	**Function**	**Parameters**
Channel Open	CreateNamedPipe	RAX: File Handler	CreateFile	RAX: File Handler
		RCX: File Name		RCX: File Name
Send	WriteFile	RCX: File Handle	WriteFile	RCX: File Handle
		RDX: Buffer Address		RDX: Buffer Address
		R9: Message Length		R9: Message Length
Receive	ReadFile	RCX: File Handle	ReadFile	RCX: File Handle
		RDX: Buffer Address		RDX: Buffer Address
		R9: Message Length		R9: Message Length
Channel Close	CloseHandle	RCX: File Handler	CloseHandle	RCX: File Handler

Table 3.3: Function List for Asynchronous Named Pipe

Event	**Server Endpoint**		**Client Endpoint**	
	Function	**Parameters**	**Function**	**Parameters**
Channel Open	CreateNamedPipe	RAX: File Handler	CreateFile	RAX: File Handle
		RCX: File Name		RCX: File Name
Send	WriteFile	RCX: File Handle	WriteFile	RCX: File Handle
		RDX: Buffer Address		RDX: Buffer Address
		R9: Message Length		R9: Message Length
Receive	ReadFile	RAX: File Handle	ReadFile	RCX: File Handle
		RDX: Buffer Address		RDX: Buffer Address
		R9: Message Length		R9: Message Length
Receive	GetOverlappedResult	RCX: File Handler	GetOverlappedResult	RCX: File Handler
		RDX: Overlap Structure address		RDX: Overlap Structure Address
Channel Close	CloseHandle	RCX: File Handler	CloseHandle	RCX: File Handler

A named pipe server is responsible for the creation of the pipe, while clients can connect to the pipe after it was created. The creation and connection of a named pipe returns the handle ID of that pipe. These handler Ids will be used later when data is being sent or received to a specified pipe. Figure3.6 shows the channel set up process for a Named Pipe communication.

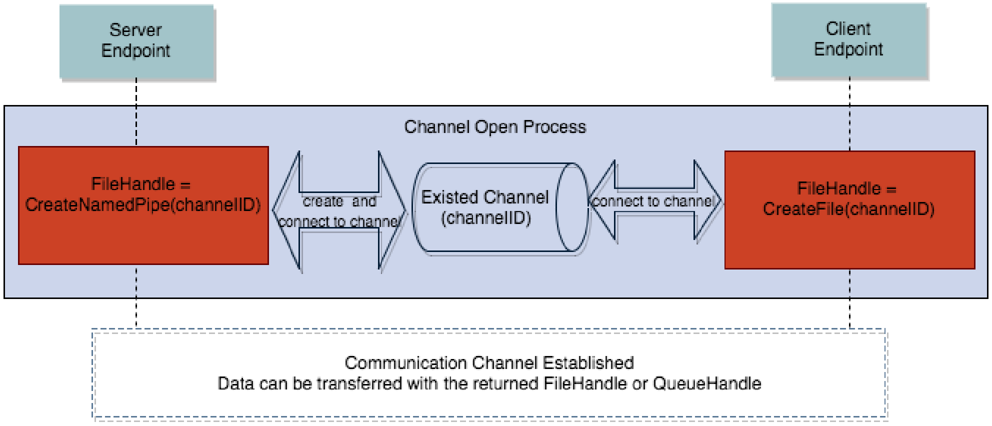

Server Endpoint
Client Endpoint
Channel Open Process
FileHandle = CreateNamedPipe(channelID)
create and connect to channel
Existed Channel (channelID)
connect to channel
FileHandle = CreateFile(channelID)
Communication Channel Established
Data can be transferred with the returned FileHandle or QueueHandle

Figure 3.6: Channel Open Process for a Named Pipe

3.4.3 Message Queue

Similar to Named Pipe, Message Queue's implementation in Windows also has two modes, synchronous and asynchronous. Moreover, the asynchronous mode further divides into two operations, one with callback function while the other without. With the callback function, the callback function would be called when the send or receive operations finish. Without callback function, the general function *MQGetOverlappedResult* should be called by the endpoints to check if the message sending or receiving operation finish, the output message will be stored in the overlap structure whose memory address saved in the function's output parameter Overlap Structure Address. Table3.4 lists the functions for synchronous mode while Table3.5 and Table3.6 list the functions for the asynchronous mode with and without callback.

Table 3.4: Function List for Synchronous MSMQ

Event	**Function**	**Parameters**
Channel Open	MQOpenQueue	RAX: Queue Handler
		RCX: Queue Format Name
Send	MQSendMessage	RCX: Queue Handle
		RDX: Message description structure Address
Receive	MQReceiveMessage	RCX: Queue Handle
		R9: Message description structure Address
Channel Close	MQCloseQueue	RCX: Queue Handler

Table 3.5: Function List for Asynchronous MSMQ with Callback

Event	**Function**	**Parameters**
Channel Open	MQOpenQueue	RAX: Queue Handler
		RCX: Queue Format Name
Send	MQSendMessage	RCX: Queue Handle
		RDX: Message description structure Address
Receive	MQReceiveMessage	RCX: Queue Handle
		R9: Message description structure Address
Receive	CallbackFuncName	Parameters for the callback function.
Channel Close	MQCloseQueue	RCX: Queue Handler

Table 3.6: Function List for Asynchronous MSMQ without Callback

Event	**Function**	**Parameters**
Channel Open	MQOpenQueue	RAX: Queue Handler
		RCX: Queue Format Name
Send	MQSendMessage	RCX: Queue Handle
		RDX: Message description structure Address
Receive	MQReceiveMessage	RCX: Queue Handle
		R9: Message description structure Address
Receive	MQGetOverlappedResult	RCX: Overlap Structure address
Channel Close	MQCloseQueue	RCX: Queue Handler

The endpoints of the communication can create the queue or use the existing one. However, both of them have to open the queue before they access it. The handle ID returned by the open queue function will be used later on when messages are being sent or received to identify the queue. Figure3.7 shows the channel set up process for a Message Queue communication.

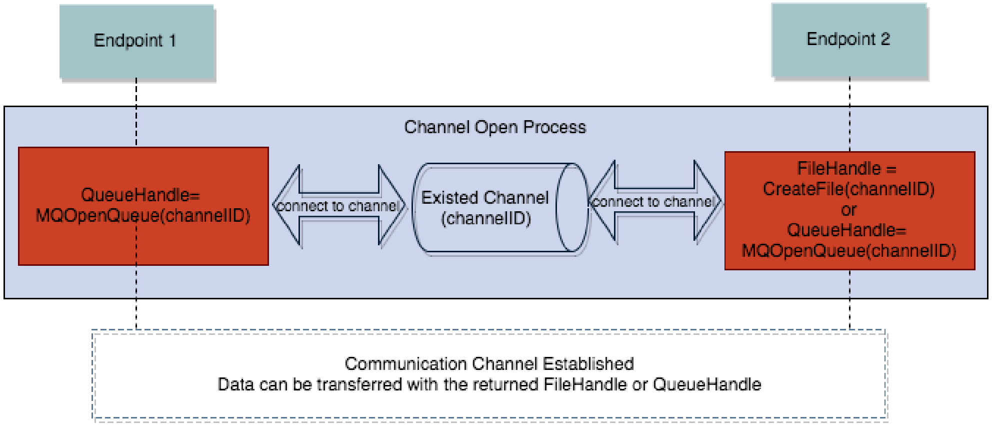


Figure 3.7: Channel Open Process for a Message Queue

3.4.4 TCP and UDP

In Windows programming, these two methods shared the same set of APIs regardless the input parameter values and operation behaviour are different. In Windows socket solution, one of the two endpoints is the server while the other one is the client. Table 3.7 lists the functions of a UDP or TCP communication.

Table 3.7: Function APIs for TCP and UDP

Event	Server Endpoint		Client Endpoint	
	Function	**Parameters**	**Function**	**Parameters**
Channel Open	socket	RAX: Socket Handle	socket	RAX: Socket Handle
Channel Open	bind	RCX: Socket Handle	connect	RCX: Socket Handle
		RDX: Server Address & Port		RDX: Server Address & Port
Channel Open	accept	RAX: New Socket Handle		
		RCX: Socket Handle		
		RDX: Client Address & Port		
Send	send	RCX: New Socket Handle	send	RCX: Socket Handle
		RDX: Buffer Address		RDX: Buffer Address
Receive	recv	RCX: New Socket Handle	recv	RCX: Socket Handle
		RDX: Buffer Address		RDX: Buffer Address
Channel Close	closesocket	RCX: New Socket Handle	closesocket	RCX: Socket Handle

The communication channel is set up by both of the endpoints. The function *socket* should be called to create their own socket on both endpoints. After the sockets are created, the server endpoint binds the socket to its service address and port by calling the function *bind*. Then the server endpoint calls the function *accept* to accept the client connection. The client will call the function *connect* to connect to the server. When the function *accept* return successfully, a new socket handle will be generated and returned for further data transfer between the server endpoint and the connected client endpoint. After all these operations are performed successfully, the channel is established and the data transfer can start. During the channel open stage, server endpoint has two socket handles, the first one is used to listen to the connection from the client, while the second one is created for real data transfer. Figure3.8 shows the channel open process for TCP and UDP.

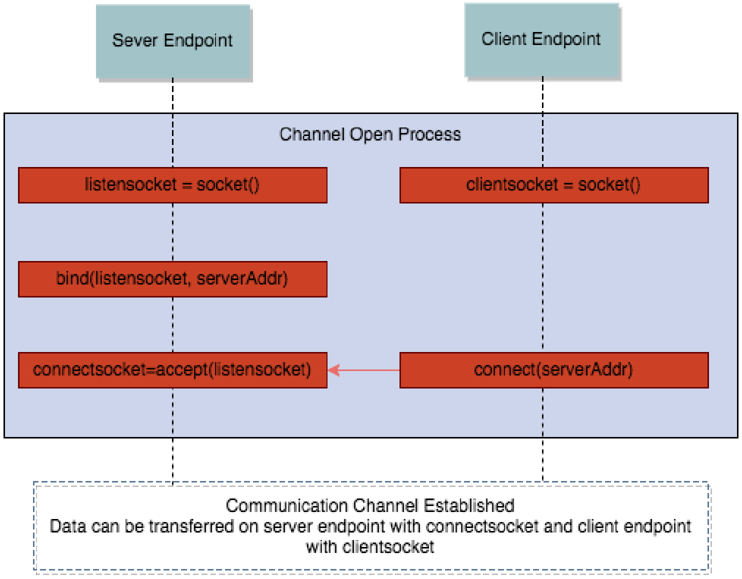

Sever Endpoint
Client Endpoint
Channel Open Process
listensocket = socket()
clientsocket = socket()
bind(listensocket, serverAddr)
connectsocket=accept(listensocket)
connect(serverAddr)
Communication Channel Established
Data can be transferred on server endpoint with connectsocket and client endpoint with clientsocket

Figure 3.8: Channel Open Model for TCP and UDP

3.5 Communication Event Locating in Assembly Execution Traces

The goal of this research is to identify the communications from the dual-trace and present them to the user. The identification of all events of the communications from the execution trace would be the first step. Last section concludes communication events including system function names, parameters and the function calls relationship. This section will 1) discuss the basis of the execution traces. and 2) conclude how to retrieve the event information from the execution traces.

3.5.1 Assembly Execution Trace

The dual-trace being analysed are in assembly level. One dual-trace contains two execution traces. There is no timing information of these two traces which means we don't know the timestamps of the events of these two traces and can not match the events from both sides by time sequence. However the captured instructions in the trace are ordered in execution sequence. The execution traces contain all executed instructions as well as the corresponding changed memory by each instruction. Additionally, system calls are also captured by instruction id, which means if .dll or .exe files are provided, the system function calls can be identified with function names. Memory states can be reconstructed from the recorded memory changes to get the data information of the communication.

3.5.2 Event Information Retrieval

For simplification, each function call is treated as an event. A function call in the trace starts from the function call instruction to the function return instruction. The input parameters' value and input buffer content should be retrieved from the memory state of the the function call instruction line while the return value, output parameters' value and output buffer content should be retrieved from the memory state of the function return instruction line. Tables in section 3.4 indicate all the functions of the communication methods as well as the concerned parameters. Following the windows calling convention, the concerned parameter value or buffer address can be found in the corresponding register or stack positions. The buffer content can be found in the memory address in the reconstructed memory state.

Chapter 4

Communication Identification Algorithms and Prototype Implementation

In the Modeling Chapter, I elaborate the definition of the communications and the strategy of the communication identification. There are several major step in the identification strategy and some of them are not strange forward. In this chapter, I will develop the algorithm for some steps in the identification strategy. A prototype of communication identification feature build on top of Atlantis[4] will also be illustrated.

4.1 Algorithms for Communication Identification in Dual-trace

4.1.1 Event Locating Algorithm

The concerned event types in a communication consist of channel open, channel close, send and receive events. Each of these event type would contain one or more functions. The functions list for a communication method is needed as a input of this algorithm. Tables in Section3.4 give the examples of event type list of communication methods. The algorithm is designed for locating events of one communications method. If more than one communication methods are being investigated, this algorithm should be run multiple times, each for a method. Events in the output event list is sorted by time of occurrence, since the search of event is line by line of the assembly instructions.

Algorithm 1: Event Locating Algorithm

```
Input: dual-trace, function list for concerned events
Output: two event lists for two separative traces in the dual-trace
$eventLists \leftarrow Map\langle String, List\langle Event\rangle\rangle$;
for $trace \in dual\text{-}trace$ do
    $eventList \leftarrow List\langle Event\rangle$;
    while not at end of trace do
        for $f \in functionList$ do
            if Is function call of $f$ then
                find function return instruction line;
                $event.inputs \leftarrow$ get input parameters from this instruction line;
                $event.outputs \leftarrow$ get output parameters and return value from the return
                  instruction line;
                $event.type \leftarrow f.eventType$;
                $eventList.add\,(event)$;
    $eventLists.add\,(eventList)$;
return $eventLists$;
```

4.1.2 Endpoint and Streams Identification Algorithm

The events located in the traces may correspond to different endpoints, the next step is to group them for each endpoints and furthermore group them into streams of the endpoints. The input of this algorithm is one of the event list from the Event Locating Algorithm. So this algorithm should be run separately for both trace in the dual-trace. Since the input event list is sorted by time of occurrence and the channel open events should always happen before other events, it is reasonable to assume the new endpoint can be identified by its first channel open function call. The output of this algorithm is the endpoint list. Each endpoint in this list contains the stream which consist of the sub streams. The concepts of the stream and sub streams are defined in Section3.1.1.

Algorithm 2: Endpoint Indentification Algorithm

```
Input: event list of a trace
Output: endpoint list
$endpoints \leftarrow Map\langle String, List\langle EndPoint\rangle\rangle$;
for $event \in eventList$ do
  –
return $endpoints$;
```

After locating all the events from the t

4.1.3 Channel Identification Algorithm

4.1.4 Transported Stream Data Verification

4.1.5 Transported Event Data Verification

4.1.6 Data Structure for Identified Communications

- Locate all events
- Identify the endpoints from the channel open events
- Group the related events into streams, send streams and receive streams, then bind them to the corresponding endpoint
- Match the endpoints to the communications
- For reliable communication, verify if the send data union in the stream of one endpoint match the receive data union in the stream of the other endpoint; For unreliable communication, try to match the send and receive events of both endpoints
- Construct and organize all the retrieved data of the communication
- Present the identified communications to the user

From the data transfer model of TCP channel, we can see that the sent packages can be shuffled in the receiver side. So it's meaningless to do a one-to-one send/receive matching from both ends of the channel. This algorithm search thought out the dual-trace, by matching the local and remote ip/port in the create, bind and connect function calls of the socket, identifies all channels of the dual-trace. For each channel, the send and receive function calls are catch. This algorithm will output all the recognized TCP channels. The output data structure of the TCP channel is list in Algorithm3, while the algorithm is list in Algorithm4, Algorithm5 and Algorithm6.

Algorithm 3: Output Data Structure for TCP Channel

```
struct {
    SocketSR trace1
    SocketSR trace2
} Channel
struct {
    Socket            socket
    List⟨ Function⟩   sends
    List⟨ Function⟩   receives
} SocketSR
struct {
    Int        handle
    String     local
    String     remote
    Function   create
    Function   bind
    Function   connect
    Function   close
} Socket
struct {
    Int        lineNum
} Function
```

Algorithm 4: TCP Channel Information Retrival Algorithm

```
Input: Trace1 and Trace2 from both sides of dual-trace
Output: All TCP channels
$channels \leftarrow List\langle Channel\rangle$
$trace1sockets \leftarrow searchSockets\,(trace1)$
$trace2sockets \leftarrow searchSockets\,(trace2)$
for $s1 \in trace1sockets$ do
    for $s2 \in trace2sockets$ do
        if $s1.local = s2.remote$ AND $s2.local = s1.remote$ then
            $channel.trace1.socket \ \ \leftarrow s1$
            $channel.trace2.socket \ \ \leftarrow s2$
            $channels.add\,(channel)$
$findAllSendAndRecv\,(trace1, channels, True)$
$findAllSendAndRecv\,(trace2, channels, Flase)$
```

Algorithm 5: searchSockets() Function for TCP Channel Information Retrival Algorithm

```
Function searchSockets(trace):
    $sockets \leftarrow Map\langle Int, Socket\rangle$
    while not at end of trace do
        if socket create function call then
            $socket.handle \leftarrow$ return value of the function call
            $socket.create.lineNum \leftarrow currentline$
            $sockets.add\,(handle, socket)$
        else if socket bind function call then
            $handle \leftarrow$ handle parameter of the function call
            $sockets[handle].local \leftarrow$ address and port parameter of the function call
            $sockets[handle].bind \leftarrow$ currentline
        else if socket connect function call then
            $handle \leftarrow$ handle parameter of the function call
            $sockets[handle].remote \leftarrow$ address and port parameter of the function call
            $sockets[handle].connect \leftarrow$ currentline
        else if socket close function call then
            $handle \leftarrow$ handle parameter of the function call
            $sockets[handle].close \leftarrow$ currentline
    for $s \in sockets$ do
        if $s.local = null$ OR $s.remote = null$ then
            $sockets.delete\,(s)$
    return $sockets.tolist()$
```

Algorithm 6: findAllSendAndRecv() Function for TCP Channel Information Retrival Algorithm

```
Function findAllSendAndRecv($trace, channels, isTrace1$):
    while not at end of trace do
        if socket send function call then
            $handle \leftarrow$ handle parameter of the function call
            $sr \leftarrow getSocketSR\,(handle, channels, isTrace1)$
            if $sr! = null$ AND $sr.socket.create.lineNum < currentline$ AND
              $sr.socket.bind.lineNum < currentline$ AND
              $sr.socket.connect.lineNum < currentline$ AND
              $sr.socket.close.lineNum > currentline$ then
                $send.lineNum \leftarrow currentline$
                $sr.sends.add(send)$
        if socket receive function call then
            $handle \leftarrow$ handle parameter of the function call
            $sr \leftarrow getSocketSR\,(handle, isTrace1)$
            if $sr! = null$ AND $sr.socket.create.lineNum < currentline$ AND
              $sr.socket.bind.lineNum < currentline$ AND
              $sr.socket.connect.lineNum < currentline$ AND
              $sr.socket.close.lineNum > currentline$ then
                $receive.lineNum \leftarrow currentline$
                $sr.receives.add(send)$
Function getSocketSR($handle, channels, isTrace1$):
    for $c \in channels$ do
        if $isTrace1$ then
            $sr \leftarrow channels.trace1$
        else
            $sr \leftarrow channels.trace2$
        if $sr.socket.handle = handle$ then
            return $sr$
```

4.2 Channel Information Retrieval Algorithm for UDP

The main difference between the UDP and TCP data transfer model is that, a UDP packet sent in the sender side always arrives as the same packet receiver side. However, the packet sent can loss and out of order. So it's meaningful to do a one-to-one send/receive matching from both ends of the channel. This algorithm applies the same search mechanism for channels as TCP, which is list in 5. However, for each channel, the catch send and receive function calls need to be matched. This algorithm will output all the recognized UDP channels. Each channel contains all matching send/receive function call pairs. The output data structure of the UDP channel is list in Algorithm7, while the algorithm is list in Algorithm8 and Algorithm9.

Algorithm 7: Output Data Structure for UDP Channel

```
struct {
    Socket trace1
    Socket trace2
    List⟨ SRPair⟩ trace1Totrace2
    List⟨ SRPair⟩ trace2Totrace1
} Channel
struct {
    Int         handle
    String      local
    String      remote
    Function    create
    Function    bind
    Function    connect
    Function    close
} Socket
struct {
    Function send
    Function recv
} SRPair
struct {
    Int         lineNum
    String      bytes
    Socket         socket
} Function
```

Algorithm 8: UDP Channel Information Retrival Algorithm

```
Input: Trace1 and Trace2 from both sides of dual-trace
Output: All UDP channels
$channels \leftarrow List\langle Channel\rangle$
$trace1sockets \leftarrow searchSockets\,(trace1)$
$trace2sockets \leftarrow searchSockets\,(trace2)$
for $s1 \in trace1sockets$ do
    for $s2 \in trace2sockets$ do
        if $s1.local = s2.remote$ AND $s2.local = s1.remote$ then
            $channel.trace1Socket \;\; \leftarrow s1$
            $channel.trace2Socket \;\; \leftarrow s2$
            $channels.add\,(channel)$
$findAllSendAndRecv\,(trace1, trace2, channels)$
Function findAllSendAndRecv($trace1, trace2, channels$):
    $trace1Sends, trace2Sends, trace1Receives, trace2Receives \leftarrow List\langle Function\rangle$
    while not at end of trace1 do
        if socket send function call then
            $addToList\,(trace2Sends, False)$
        if socket receive function call then
            $addToList\,(trace2Receives, False)$
    while not at end of trace2 do
        if socket send function call then
            $addToList\,(trace2Sends, False)$
        if socket receive function call then
            $addToList\,(trace2Receives, False)$
    for $s \in trace1Sends$ do
        for $r \in trace2Receives$ do
            if $s.bytes = r.bytes$ then
                $SRPair.send \leftarrow s$
                $SRPair.recv \leftarrow r$
    for $s \in trace2Sends$ do
        for $r \in trace1Receives$ do
            if $s.socket.local = r.socket.remote$ AND $r.socket.local = s.socket.remote$
              AND $s.bytes = r.bytes$ then
                $SRPair.send \leftarrow s$
                $SRPair.recv \leftarrow r$
```

Algorithm 9: Other Functions for UDP Channel Information Retrival Algorithm

```
Function addToList($List, isTrance1$):
    $handle \leftarrow$ handle parameter of the function call
    $sr \leftarrow getSocketSR(handle, isTrace1)$
    if $sr! = null$ AND $sr.socket.create.lineNum < currentline$ AND
      $sr.socket.bind.lineNum < currentline$ AND
      $sr.socket.connect.lineNum < currentline$ AND
      $sr.socket.close.lineNum > currentline$ then
        $func.lineNum \leftarrow currentline$
        $func.bytes \leftarrow$ data that received when the function return
        $func.socket \leftarrow sr$
        $list.add(func)$
Function getSocketSR($handle, channels, isTrace1$):
    for $c \in channels$ do
        if $isTrace1$ then
            $sr \leftarrow channels.trace1$
        else
            $sr \leftarrow channels.trace2$
        if $sr.socket.handle = handle$ then
            return $sr$
Function getChannel($channels, trace1Socket, trace2Socket$):
    for $c \in channels$ do
        if $trace1Socket = c.trace1$ AND $trace2Socket = c.trace2$ then
            return $c$
```

4.3 Channel Information Retrieval Algorithm for Named pipe

Similar to the data transfer model of TCP channel, the sent packages can be shuffled in the receiver side in Named pipe. So we don't do one-to-one send/receive matching for Named pipe channel as neither. The channel search is based only on the name of the Named pipe. In both ends, the name for the Name pipe is identical but have different handles. For each channel, the send and receive function calls are catch. This algorithm will output all the recognized Named pipe channels. The output data structure of the channel is list in Algorithm10, while the algorithm is list in Algorithm11, Algorithm**??** and Algorithm12.

Algorithm 10: Output Data Structure for Named pipe Channel

```
struct {
    pipeSR trace1
    pipeSR trace2
} RebuiltChannel
struct {
    pipeEnd            end
    List⟨ Function⟩    sends
    List⟨ Function⟩    receives
} pipeSR
struct {
    Int         handle
    String      pipeName
    Function    create
    Function    close
} pipeEnd
struct {
    Int         lineNum
} Function
```

Algorithm 11: Named Pipe Channel Information Retrival Algorithm

```
Input: Trace1 and Trace2 from both sides of dual-trace
Output: All Named Pipe channels
$channels \leftarrow List\langle Channel\rangle$
$trace1PipeEnds \leftarrow searchPipeEnds\,(trace1)$
$trace2PipeEnds \leftarrow searchPipeEnds\,(trace2)$
for $e1 \in trace1PipeEnds$ do
    for $e2 \in trace2PipeEnds$ do
        if $e1.pipeName = e2.pipeName$ then
            $channel.trace1Socket \ \leftarrow e1$
            $channel.trace2Socket \ \leftarrow e2$
            $channels.add\,(channel)$
$findAllSendAndRecv\,(trace1, channels, True)$
$findAllSendAndRecv\,(trace2, channels, Flase)$
Function searchPipeEnds($trace$):
    $ends \leftarrow Map\langle Int, pipeEnd\rangle$
    while not at end of trace do
        if Name create function call then
            $end.handle \leftarrow$ return value of the function call
            $end.pipeName \leftarrow$ pipName parameter of the function call
            $end.create.lineNum \leftarrow currentline$
            $ends.add\,(handle, socket)$
        else if socket close function call then
            $handle \leftarrow$ handle parameter of the function call
            $ends[handle].close \leftarrow$ currentline
    return $sockets.tolist()$
```

Algorithm 12: findAllSendAndRecv() Function for Named Pipe Channel Information Retrival Algorithm

```
Function findAllSendAndRecv($trace, channels, isTrace1$):
    while not at end of trace do
        if pipe send function call then
            $handle \leftarrow$ handle parameter of the function call
            $sr \leftarrow getEndSR\,(handle, channels, isTrace1)$
            if $sr! = null$ AND $sr.end.create.lineNum < currentline$ AND
              $sr.end.close.lineNum > currentline$ then
                $send.lineNum \leftarrow currentline$
                $sr.sends.add(send)$
        if pipe receive function call then
            $handle \leftarrow$ handle parameter of the function call
            $sr \leftarrow getEndSR\,(handle, isTrace1)$
            if $sr! = null$ AND $sr.socket.create.lineNum < currentline$ AND
              $sr.socket.close.lineNum > currentline$ then
                $receive.lineNum \leftarrow currentline$
                $sr.receives.add(send)$

Function getEndSR($handle, channels, isTrace1$):
    for $c \in channels$ do
        if $isTrace1$ then
            $endSr \leftarrow channels.trace1$
        else
            $endSr \leftarrow channels.trace2$
        if $sr.socket.handle = handle$ then
            return $endSr$
```

4.4 Communication Identification Feature Prototype

Chapter 5

Feature Prototype On Atlantis

In this section we discuss the design of the prototype of dual-trace analysis. The tool prototype I built was based on the general communication model I described in last section. This prototype consist of three main components: user interface for defining the communication type, algorithm of locating the communication events in the dual-trace, user interface and strategy to navigate the located events to the sender and receiver traces. We provide the background information of the design of each component as well as their detail design in each corresponding subsection.

5.0.1 User Defined Channel Type By Json

In our design, we don't specify any predefined communication type but give the user ability to do that. By the user interface implemented, the user can defined their own communication type. This give the flexibility to the user to define what they are looking for. Each communication type consist of 4 system function calls. They are channel create/open in sender and receiver sides, sender's send message function and receiver's receive message function. By indicating the channel create/open functions in both sender and receiver sides, the tool can acquire the channel's identifiers. Later on the tool can match the send and received messages within a specific channel. The send and receive functions are used to located the event happened in the traces. The messages sent and received are reconstructed from the memory state when the send and receive functions are called and returned. The detail of the match algorithm will be discuss later.

Channel Searching

The called functions' name can be inspected by search of the symbolic name in the executable binary or any DLLs which used by the program at the time when it is traced. This functionality exists in the current Atlantis. By importing the DLLs and execution executable binary, Atlantis can list all called functions for the users in the Functions view. From this list, users can chose the interested functions and generate their interested communication type. In Figure5.1 there is an action item ”Add to Communication type” in the right click menu of the function entry. Figure 5.2 shows the dialogue for entering the information for the adding function. As this figure shows, users can get the existing communication type list in the drop down menu. They can choose to add the current function to an exist communication type or they can add it to a new communication type by entering a new name. For the channel create/open function, the register holding the address of channel’s name as input and the register holding the handle identification of the channel as output are required. For the send/receive function, the register holding the address of the send/receiver buffer, the register holding the length of the sending/receiving message and the register holding the channel’s identification are required. As there are 4 functions for each communication type users have to repeat this add function to communication type action for 4 times to generate one communication type.

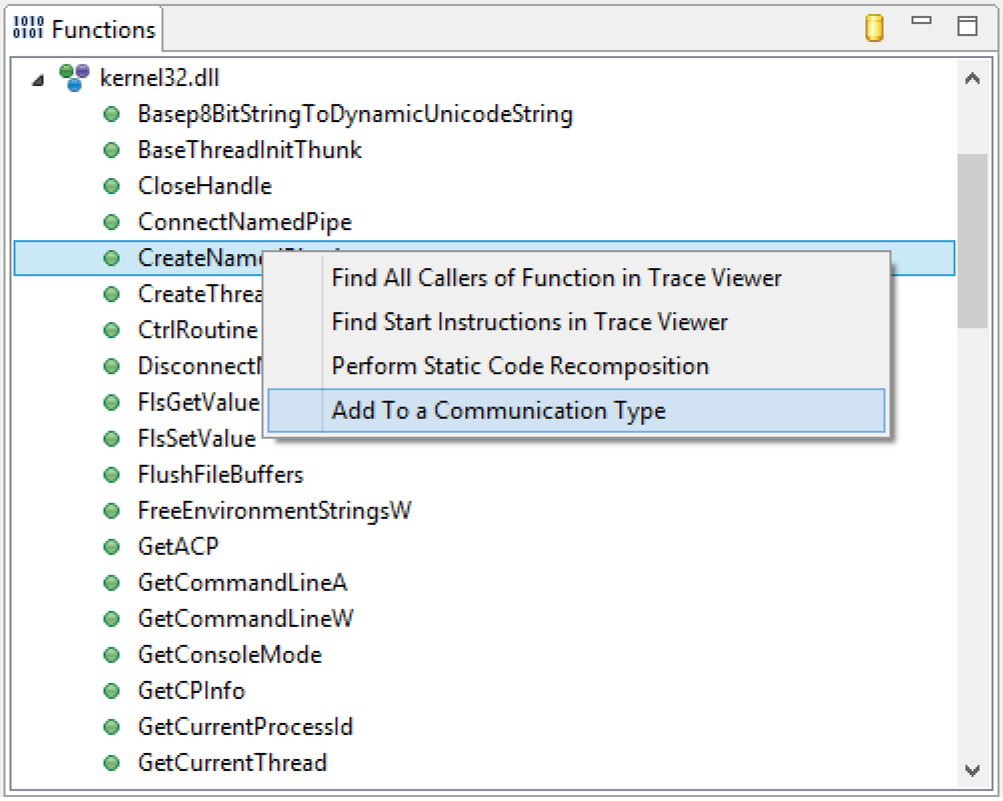
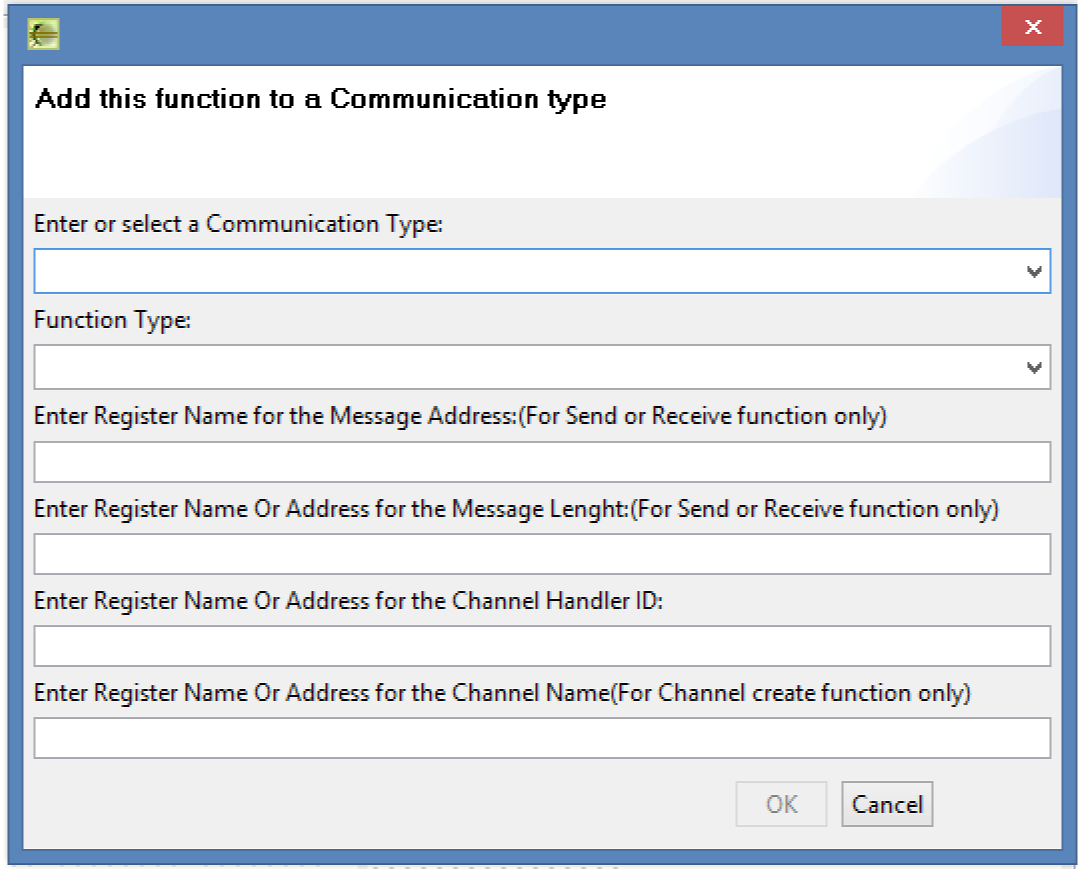


Figure 5.1: Add function to a Communication type from Functions View



Add this function to a Communication type

Enter or select a Communication Type:

Function Type:

Enter Register Name for the Message Address:(For Send or Receive function only)

Enter Register Name Or Address for the Message Lenght:(For Send or Receive function only)

Enter Register Name Or Address for the Channel Handler ID:

Enter Register Name Or Address for the Channel Name(For Channel create function only)

Figure 5.2: Dialog to input information for a function adding to a communication type

Communication Type Data Structure

The defined communication type will be stored in a xml file. The list below shows the data structure of one communication type.

```
<messageTypesData>
    <parentFolder >.tmp</parentFolder >
    <messageTypes>
        <messageType>
            <name>namedPipe_clientsend </name>
            <sendFunction >
                <associatedFileName >Client </associatedFileName >
                <name>WriteFile </name>
                <messageAddress >RDX</messageAddress >
                <messageLengthAddress >R8</messageLengthAddress >
                <channelIdReg>RCX</channelIdReg>
            </sendFunction>
            <receiveFunction>
                <associatedFileName>Server </associatedFileName>
                <name>ReadFile </name>
                <messageAddress>RDX</messageAddress>
                <messageLengthAddress>R8</messageLengthAddress>
                <channelIdReg>RCX</channelIdReg>
            </receiveFunction>
            <sendChannelCreateFunction>
                <associatedFileName>Client </associatedFileName>
                <name>CreateFileA </name>
                <channelIdReg>RAX</channelIdReg>
                <channelNameAddress>RCX</channelNameAddress>
            </sendChannelCreateFunction>
            <receiveChannelCreateFunction>
                <associatedFileName>Server </associatedFileName>
                <name>CreateNamedPipeA </name>
                <channelIdReg>RAX</channelIdReg>
                <channelNameAddress>RCX</channelNameAddress>
            </receiveChannelCreateFunction>
        </messageType>
    </messageTypes>
</messageTypesData>
```

Communication Type View

A new view named Communication Types view is for the user defined communication types. All user defined communication type are stored in the .xml file and listed in communication type view when it's opened as shown in Figure 5.3. User can change the name of a communication type, remove an existing communication type or searching of the match message occurrences of selected communication type by selecting action item in the right click menu of an communication type entry. The matched messages are listed in the result window of the view. By clicking the entry of the search result, user can navigate to it's sender or receiver's corresponding instruction line as shown in Figure5.4. Message content in the memory view will be shown as well.

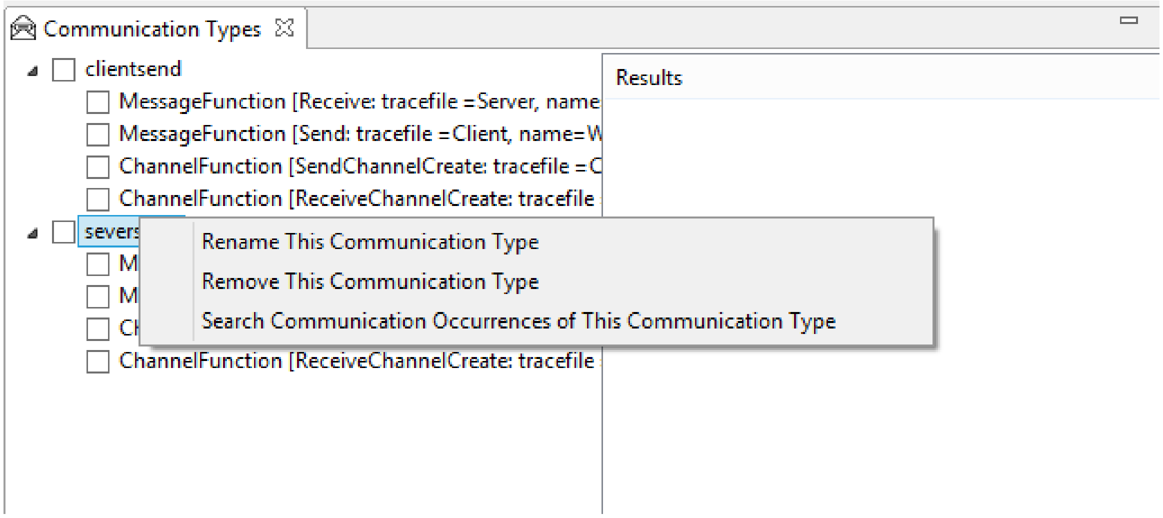

Communication Types
clientsend
MessageFunction [Receive: tracefile =Server, name
MessageFunction [Send: tracefile =Client, name=W
ChannelFunction [SendChannelCreate: tracefile =C
ChannelFunction [ReceiveChannelCreate: tracefile
severs
M
M
Ch
ChannelFunction [ReceiveChannelCreate: tracefile
Results
Rename This Communication Type
Remove This Communication Type
Search Communication Occurrences of This Communication Type

Figure 5.3: New View: Communication Type View

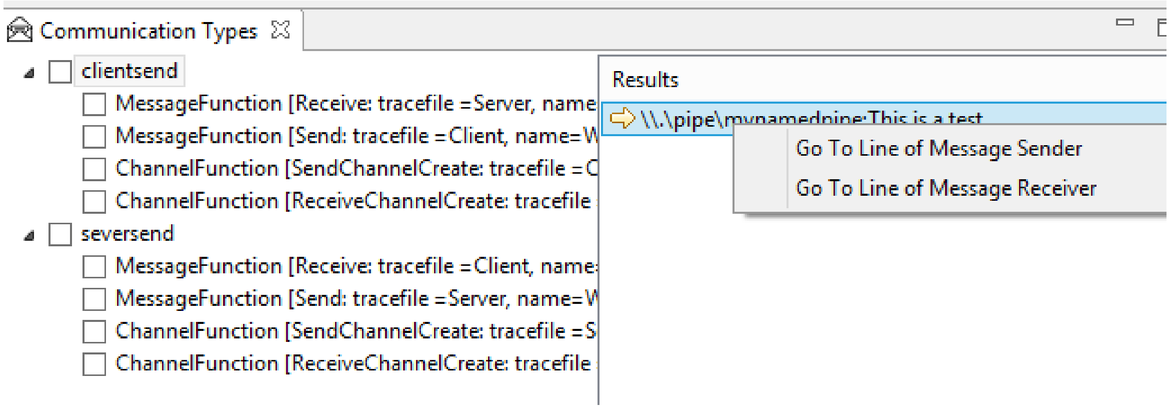

Communication Types
clientsend
MessageFunction [Receive: tracefile =Server, name
MessageFunction [Send: tracefile =Client, name=W
ChannelFunction [SendChannelCreate: tracefile =C
ChannelFunction [ReceiveChannelCreate: tracefile
seversend
MessageFunction [Receive: tracefile =Client, name
MessageFunction [Send: tracefile =Server, name=W
ChannelFunction [SendChannelCreate: tracefile =S
ChannelFunction [ReceiveChannelCreate: tracefile
Results
\\.\pipe\mynamedpipe:This is a test
Go To Line of Message Sender
Go To Line of Message Receiver

Figure 5.4: Right Click menu to navigate to send and receive event in the traces

5.0.2 Communication Event Searching

The communication event consists of the send message event in the sender side and receive message event in the receiver side. The communication event searching algorithm can be divided into three main steps: 1. search all channel create/open event in the sender and receiver side, save the handle id and corresponding channel name. 2. Search all message send and receive event in sender and receiver sides. 3. Matching the send/receive messages pair based on the channel names and message contents.

Record opened Channel

In this step the algorithm is supposed to search all the open channel both in the sender and receiver side. The found created channels are recorded in a map. The key of the map is the handler id of the channel and the value is the channel name. A channel in the sender and receiver sides will have different handler id but same channel name.

Search send and receive Message

All send message and receive message function calls will be found out in the trace. When a send function hit, the memory state of the hit instruction line will be reconstructed, and the message content can be get from the memory with the send message buffer address. When a receive function hit, the return line of that function is needed for getting the message content. The memory state of the function return line is reconstructed and the message content can be get from the reconstructed memory state with the receive message buffer address.

Matching the send/receive messages pair

After the created channel and send/receive message are found out in the sender and receiver side, a matching algorithm is used to match the send/receive message pairs.

Matching Event Data Structure

The matching event is stored in cache when the tool is running. Only the most recent search result is cached currently. If users need the previous result, they need to apply the search again. The matching Event consist of two sub-events, one is message send event while the other is message receive event. Both of these two sub-events are object of BfvFileMessageMatch. BfvFileMessageMatch is an Java class extends org.eclipse.search.internal.ui.text.FileMatch. FileMatch class containing the information needed to navigate to the trace file editor. In order to show the corresponding send/receive message in the memory view, the target memory address storing the message content is set in BfvFileMessageMatch. Two more elements: message and channel name are also set in BfvFileMessageMatch which are listed in the search result.

5.0.3 Navigation From Channel Search Result to Dual-Trace

The right click menu of an entry in the search result list has two action items: Go To Line of Message Sender and Go To Line of Message Receiver. Both of the action items allow users to navigate to the trace Instruction view. When the user click on these items, it will navigate to the corresponding trace sender or receiver trace instruction view. Meanwhile the memory view jumps to the target address of the message buffer, and the memory state is reconstructed so that the message content in that buffer will be shown in the memory view.

Chapter 6

Evaluation

The case we used to test this prototype contains one named pipe synchronous channel between a server and a client. Client send a message to the server and server reply another message to the client.

6.0.1 Experiment Verification Design

The test cases are designed to find all the messages from client to server and all the messages from server to client. Two end to end test cases are designed for both scenarios.

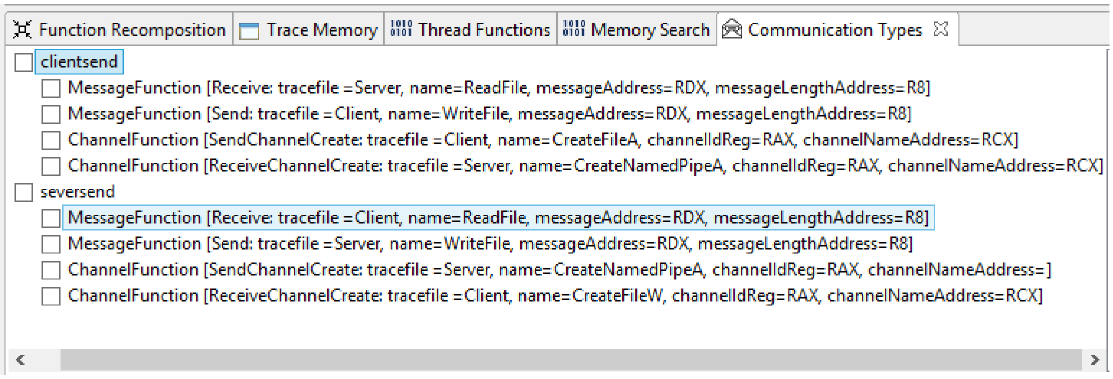
In each test case, there are three test steps: 1. define the communication type by adding channel creating functions and message send/receive functions of server and client sides. 2. search for the events of the defined communication type. 3. for the occurrence of the events, navigate to the trace instruction and memory view.

Verification points are specified for each step as: 1. verify the communication types with their functions are listed in the communication view. 2. verify the message events in the dual-trace can be found and listed in the search result view. 3. verify the navigation from the result entry to the instruction view of sender trace and receiver trace.

6.0.2 Result

We used the dual-trace provided by DRDC and follow the experiment and verification design to conduct this test. Figure6.1 shows that the user defined clientsend and serversend communication types are shown in the communication type view as well as the functions consist of the communication types. Figure6.3 shows the search result of clientsend communication type, while Figure6.3 shows the search result of the serversend communication type. By clicking the Go To Line of

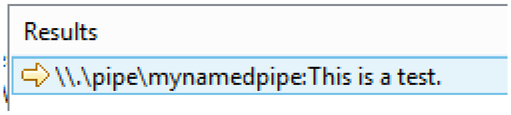
Message Sender and Go To Line of Message Receiver action items, instruction view and memory view updated correctly. Figure6.4 shows the server was sending out a message: This is an answer.



□ clientsend
- □ MessageFunction [Receive: tracefile =Server, name=ReadFile, messageAddress=RDX, messageLengthAddress=R8]
- □ MessageFunction [Send: tracefile =Client, name=WriteFile, messageAddress=RDX, messageLengthAddress=R8]
- □ ChannelFunction [SendChannelCreate: tracefile =Client, name=CreateFileA, channelIdReg=RAX, channelNameAddress=RCX]
- □ ChannelFunction [ReceiveChannelCreate: tracefile =Server, name=CreateNamedPipeA, channelIdReg=RAX, channelNameAddress=RCX]

□ seversend
- □ MessageFunction [Receive: tracefile =Client, name=ReadFile, messageAddress=RDX, messageLengthAddress=R8]
- □ MessageFunction [Send: tracefile =Server, name=WriteFile, messageAddress=RDX, messageLengthAddress=R8]
- □ ChannelFunction [SendChannelCreate: tracefile =Server, name=CreateNamedPipeA, channelIdReg=RAX, channelNameAddress=]
- □ ChannelFunction [ReceiveChannelCreate: tracefile =Client, name=CreateFileW, channelIdReg=RAX, channelNameAddress=RCX]

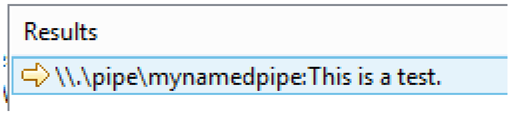
Figure 6.1: Defined clientsend and serversend communication types in Communication View



Results

\\.\pipe\mynamedpipe:This is a test.

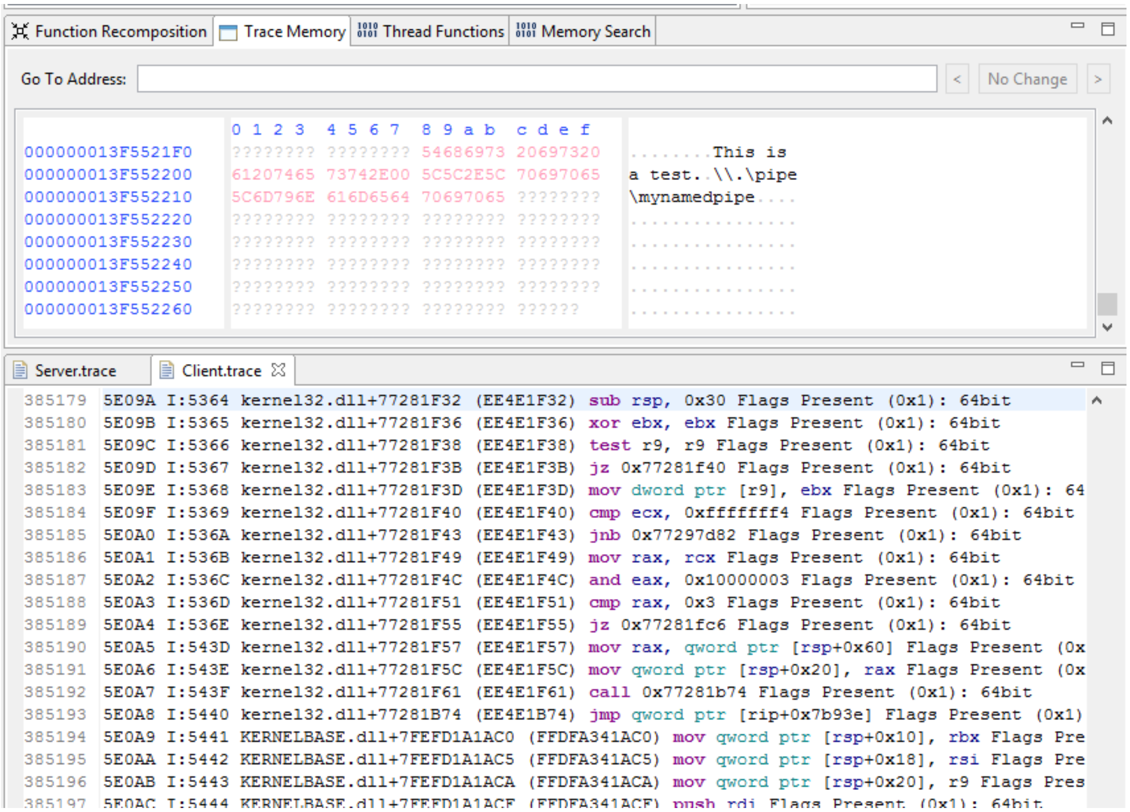
Figure 6.2: the search result of clientsend communication type



Results

\\.\pipe\mynamedpipe:This is a test.

Figure 6.3: the search result of the serversend communication type



```
                  0 1 2 3  4 5 6 7  8 9 a b  c d e f
000000013F5521F0  ???????? ???????? 54686973 20697320  ........This is
000000013F552200  61207465 73742E00 5C5C2E5C 70697065  a test..\\.\pipe
000000013F552210  5C6D796E 616D6564 70697065 ????????  \mynamedpipe....
000000013F552220  ???????? ???????? ???????? ????????  ................
000000013F552230  ???????? ???????? ???????? ????????  ................
000000013F552240  ???????? ???????? ???????? ????????  ................
000000013F552250  ???????? ???????? ???????? ????????  ................
000000013F552260  ???????? ???????? ???????? ??????    ................
```

```
5E09A I:5364 kernel32.dll+77281F32 (EE4E1F32) sub rsp, 0x30 Flags Present (0x1): 64bit
5E09B I:5365 kernel32.dll+77281F36 (EE4E1F36) xor ebx, ebx Flags Present (0x1): 64bit
5E09C I:5366 kernel32.dll+77281F38 (EE4E1F38) test r9, r9 Flags Present (0x1): 64bit
5E09D I:5367 kernel32.dll+77281F3B (EE4E1F3B) jz 0x77281f40 Flags Present (0x1): 64bit
5E09E I:5368 kernel32.dll+77281F3D (EE4E1F3D) mov dword ptr [r9], ebx Flags Present (0x1): 64
5E09F I:5369 kernel32.dll+77281F40 (EE4E1F40) cmp ecx, 0xfffffff4 Flags Present (0x1): 64bit
5E0A0 I:536A kernel32.dll+77281F43 (EE4E1F43) jnb 0x77297d82 Flags Present (0x1): 64bit
5E0A1 I:536B kernel32.dll+77281F49 (EE4E1F49) mov rax, rcx Flags Present (0x1): 64bit
5E0A2 I:536C kernel32.dll+77281F4C (EE4E1F4C) and eax, 0x10000003 Flags Present (0x1): 64bit
5E0A3 I:536D kernel32.dll+77281F51 (EE4E1F51) cmp rax, 0x3 Flags Present (0x1): 64bit
5E0A4 I:536E kernel32.dll+77281F55 (EE4E1F55) jz 0x77281fc6 Flags Present (0x1): 64bit
5E0A5 I:543D kernel32.dll+77281F57 (EE4E1F57) mov rax, qword ptr [rsp+0x60] Flags Present (0x
5E0A6 I:543E kernel32.dll+77281F5C (EE4E1F5C) mov qword ptr [rsp+0x20], rax Flags Present (0x
5E0A7 I:543F kernel32.dll+77281F61 (EE4E1F61) call 0x77281b74 Flags Present (0x1): 64bit
5E0A8 I:5440 kernel32.dll+77281B74 (EE4E1B74) jmp qword ptr [rip+0x7b93e] Flags Present (0x1)
5E0A9 I:5441 KERNELBASE.dll+7FEFD1A1AC0 (FFDFA341AC0) mov qword ptr [rsp+0x10], rbx Flags Pre
5E0AA I:5442 KERNELBASE.dll+7FEFD1A1AC5 (FFDFA341AC5) mov qword ptr [rsp+0x18], rsi Flags Pre
5E0AB I:5443 KERNELBASE.dll+7FEFD1A1ACA (FFDFA341ACA) mov qword ptr [rsp+0x20], r9 Flags Pres
5E0AC I:5444 KERNELBASE.dll+7FEFD1A1ACF (FFDFA341ACF) push rdi Flags Present (0x1): 64bit
```

Figure 6.4: instruction view and memory view updated correctly

Chapter 7

Conclusions

7.1 Limitations

In this section, we specify the limitations of the current prototype and the reasons for them.

7.1.1 Event Status: Success or Fail

In current prototype, we only consider the success cases. For the Fail case, since the message was not successfully sent or received, there are high chance that they are not existed in the memory of the trace. From the assembly level trace, if the message was not traced in the memory, there is no way to match the sent/received message pair in the trace analysis.

7.1.2 Match Events Distinguishing

Distinguishing is considered when multiple clients connecting to the same server. Each connection is considered as an instance. In the server side all this instances have the same pipe name but different handler ID. However in the assembly trace level there is no way to match a client with it instance handler ID. In consequence, if the same content messages are being sent/received by different clients, when the user want to match the message pair between a client and the server, there is no way to distinguish the correct one from the assembly trace level. As a result, our tool will list all the matched content message event, regardless if it's from the interested client. The user can distinguish the correct ones for this client, if they have extra information.

7.1.3 Match Events Ordering

Ordering is considered when multiple messages with exactly the same content being send/receive between the client and server. If the channel is synchronous, the order of the event is always consist with the order they happen in both the sender and receive sides. However for the asynchronous channel, there are chance that the sent messages in the sender side's trace are out of order with the received messages in the received side's trace. Unfortunately, There is no way in the assembly level trace to match the exactly ones. As a result, our tool can only order the event based on the order they happen in the traces.

7.1.4 Buffer Sizes Of Sender and Receiver Mismatch

In current prototype, we only consider the success cases. For the Fail case, since the message was not successfully sent or received, there are high chance that they are not existed in the memory of the trace. From the assembly level trace, if the message was not traced in the memory, there is no way to match the sent/received message pair in the trace analysis.

Appendix A

Additional Information

Bibliography

[1] B. Cleary, P. Gorman, E. Verbeek, M. A. Storey, M. Salois, and F. Painchaud. Reconstructing program memory state from multi-gigabyte instruction traces to support interactive analysis. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 42–51, October 2013.

[2] Mark Dowd, John McDonald, and Justin Schuh. *Art of Software Security Assessment, The: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional., 1st edition, November 2006.

[3] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA, 2008.

[4] Huihui Nora Huang, Eric Verbeek, Daniel German, Margaret-Anne Storey, and Martin Salois. Atlantis: Improving the analysis and visualization of large assembly execution traces. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 623–627. IEEE, 2017.

[5] Intel. Pin - A Dynamic Binary Instrumentation Tool | Intel Software.

[6] School of Computing) Advisor (Prof. B. Kang) KAIST CysecLab (Graduate School of Information Security. c0demap/codemap: Codemap.

[7] MultiMedia LLC. Named pipes (windows), 2017.

[8] Arohi Redkar, Ken Rabold, Richard Costall, Scot Boyd, and Carlos Walzer. *Pro MSMQ: Microsoft Message Queue Programming*. Apress, 2004.

[9] Chao Wang and Malay Ganai. Predicting Concurrency Failures in the Generalized Execution Traces of x86 Executables. In *Runtime Verification*, pages 4–18. Springer, Berlin, Heidelberg, September 2011. DOI: 10.1007/978-3-642-29860-8_2.