Dual Trace Communication Event Analysis

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

© Huihui Nora Huang, 2018
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Dual Trace Communication Event Analysis

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

List of Tables

List of Figures

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

Chapter 2

Methodology

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.

2.1 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.

2.2 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.

2.3 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.

2.4 Develop the Dual Trace Synchronization Algorithms

2.5 Match the Channels in WCF to the Model

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.

2.6 Implement the Trace Synchronization Algorithms

2.7 Verify the Algorithm by Traces from DRDC

Chapter 3

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.

3.1 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].

3.2 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[7].

3.3 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 [5] to provide more sufficient features for ”run-trace” visualization. PIN[4] 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

3.4 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.

3.5 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.

3.6 Assembly Calling Convention

Before we jumping into a specific communication channel, it is important to know some basic assembly calling convention. Calling Convention is different for operating system and the programming language. Since we are looking into the messaging methods being used in windows communication framework, and since our case study is running on a Microsoft* x64 system, 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.

Chapter 4

Channel Modeling

In this section, I modeled communication channels in the traces. There are two model, guaranteed communication model and in-guaranteed communication model. For both of these two model the communication consist of three stages: 1. channel opening, 2. data exchanging and 3. channel closing.

4.1 Communications Stages

A communication in this work is happen in a channel. There are 3 main stages of a communication in this model: 1. open the channel, 2. message send/receive, 3. close the channel. Figure 4.1 indicate how communications happen between two ends of both sides of a channel. In the open channel stage, each end need to call its own channel open functions. This open function might be one or more functions which can be channel create function, open function, connect function etc. In the message send/receive stage, messages are being sent into the channel in one side and received in the other side. In the final channel close stage, the channel delete function, disconnect function, close function etc. will be called. The channel can be reopen again to start new communications after the close stage. However the reopen channel will be treated as a communication cycle.

The operations being concerned in this model are: channel open in sender side, channel open in the receiver side, message send, message receive, channel close in sender side, channel close in receiver side. The number of send and receive function calls for one message do not necessary to be the same. It can be one to one, one to multiple, multiple to one, or multiple to multiple. Both sides of the channel shared the same channel name but different channel handle for its own operations.

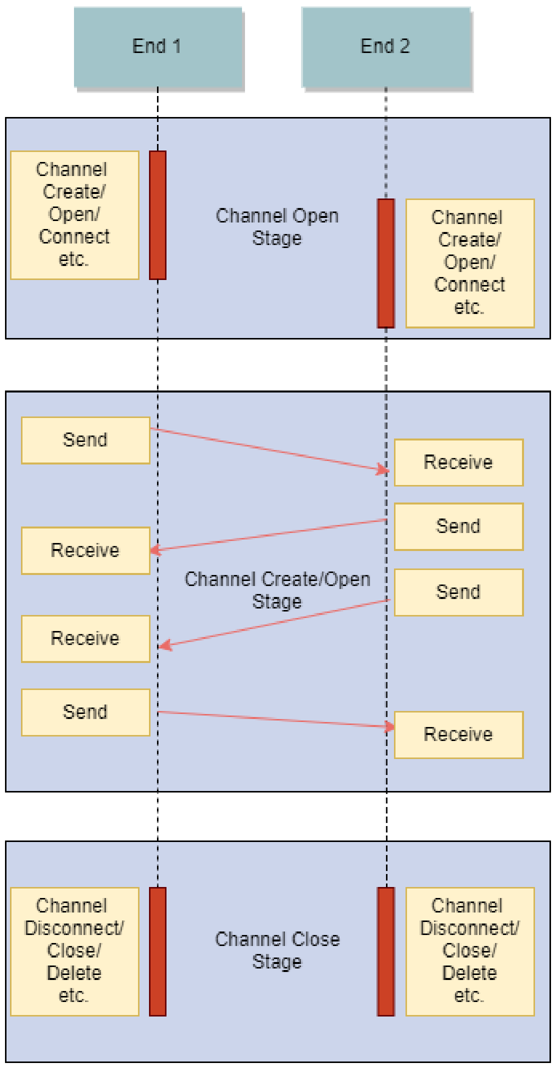

End 1
End 2
Channel Create/ Open/ Connect etc.
Channel Open Stage
Channel Create/ Open/ Connect etc.
Send
Receive
Send
Receive
Channel Create/Open Stage
Send
Receive
Send
Receive
Channel Disconnect/ Close/ Delete etc.
Channel Close Stage
Channel Disconnect/ Close/ Delete etc.

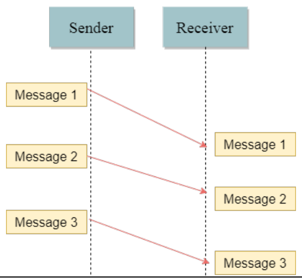
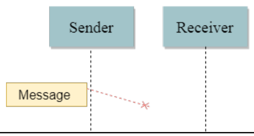
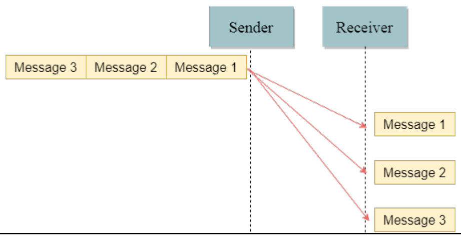
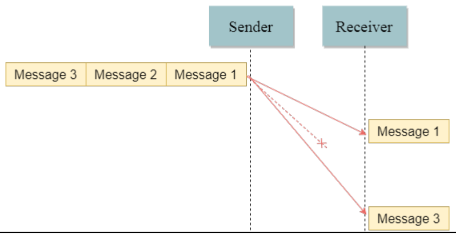
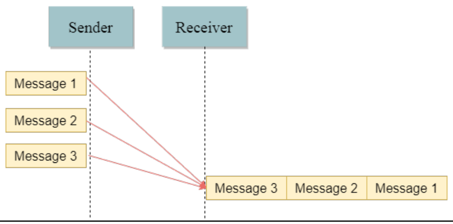
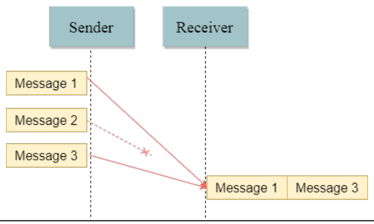
Figure 4.1: Communication Model

4.2 Communication Scenarios

Both the order guaranteed communication model and the order in-guaranteed communication model contains several send/receive scenarios. Table 4.1 shows all of send/receive scenarios for guaranteed communication model while Table 4.2 shows all of send/receive scenarios for in-guaranteed communication model.

Table 4.1: Send/Receive Scenarios of Order Guaranteed Communication Model

Scenario	**Diagram**	**Comment**

1	 Sender Receiver Message 1 Message 2 Message 3 Message 1 Message 2 Message 3	Message send/receive successfully in order
3		Message send/receive fail
4	 Sender Receiver Message 3 Message 2 Message 1 Message 1 Message 2 Message 3	One sent message received segmented
6	 Sender Receiver Message 3 Message 2 Message 1 Message 1 Message 3	One sent message received segmented in order, but partially lost
8	 Sender Receiver Message 1 Message 2 Message 3 Message 3 Message 2 Message 1	Multiple messages received at a time
9	 Sender Receiver Message 1 Message 2 Message 3 Message 1 Message 3	Multiple messages received at a time, but partially lost

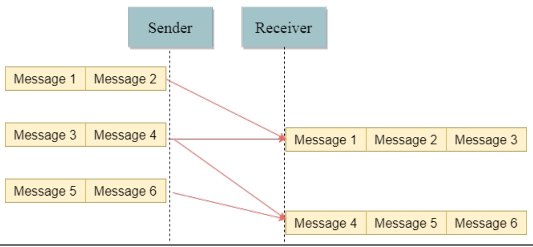
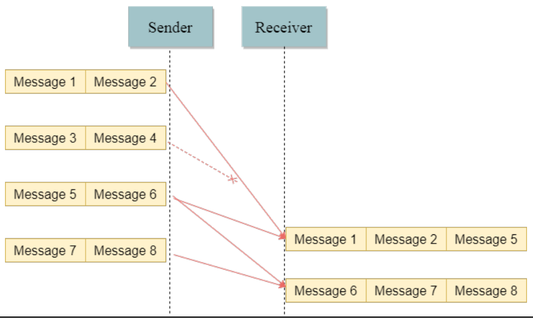
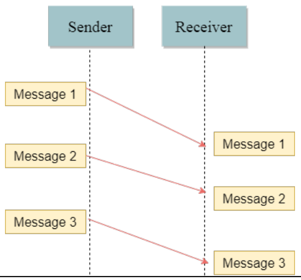
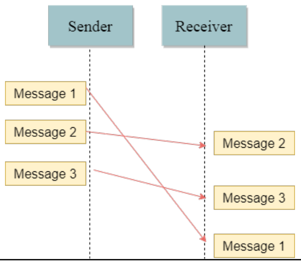
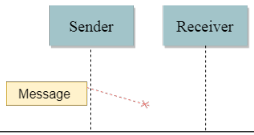
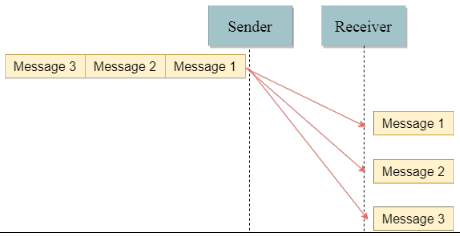
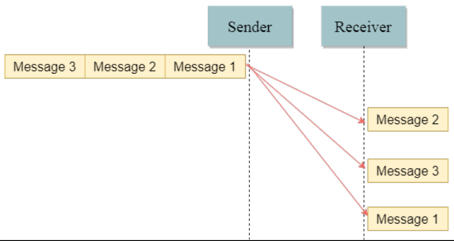
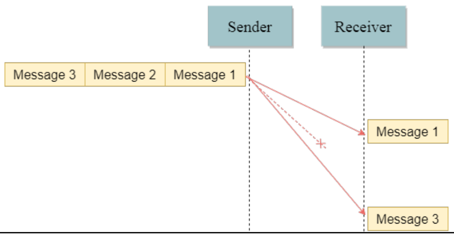
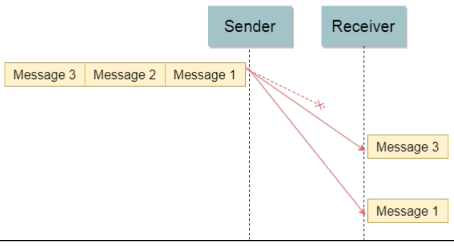
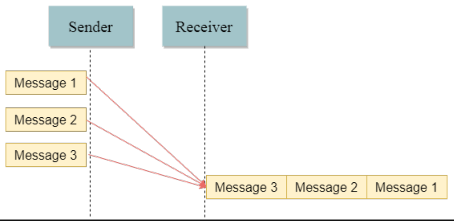
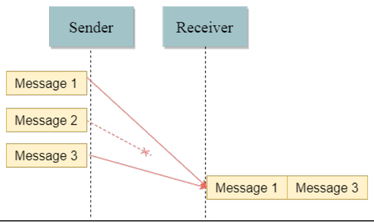
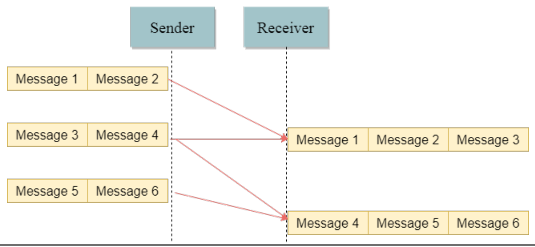
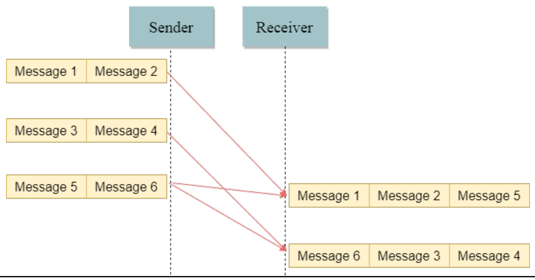
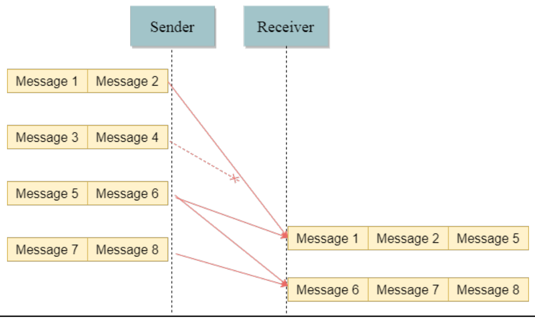
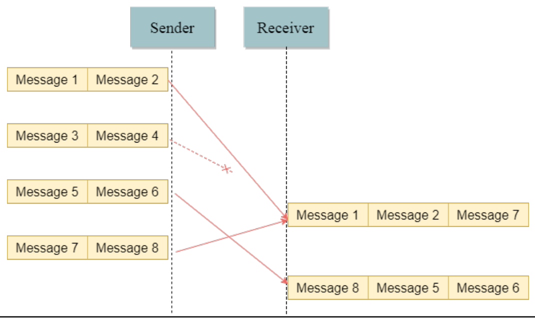
10	 Sender Receiver Message 1 Message 2 Message 3 Message 4 Message 5 Message 6 Message 1 Message 2 Message 3 Message 4 Message 5 Message 6	Messages segmented and reconstructed in order
12	 Sender Receiver Message 1 Message 2 Message 3 Message 4 Message 5 Message 6 Message 7 Message 8 Message 1 Message 2 Message 5 Message 6 Message 7 Message 8	Messages segmented and reconstructed partially lost

Table 4.2: SSend/Receive Scenarios of Order In-Guaranteed Communication Model

Scenario	Diagram	Comment
1	 Sender Receiver Message 1 Message 2 Message 3 Message 1 Message 2 Message 3	Message send/receive successfully in order
2	 Sender Receiver Message 1 Message 2 Message 3 Message 2 Message 3 Message 1	Message send/receive successfully but out of order
3		Message send/receive fail

4	 Sender Receiver Message 3 Message 2 Message 1 Message 1 Message 2 Message 3	One sent message received segmented
5	 Sender Receiver Message 3 Message 2 Message 1 Message 2 Message 3 Message 1	One sent message received segmented and out of order
6	 Sender Receiver Message 3 Message 2 Message 1 Message 1 Message 3	One sent message received segmented in order, but partially lost
7	 Sender Receiver Message 3 Message 2 Message 1 Message 3 Message 1	One sent message received segmented out of order and partially lost
8	 Sender Receiver Message 1 Message 2 Message 3 Message 3 Message 2 Message 1	Multiple messages received at a time
9	 Sender Receiver Message 1 Message 2 Message 3 Message 1 Message 3	Multiple messages received at a time, but partially lost

10	 Sender Receiver Message 1 Message 2 Message 3 Message 4 Message 5 Message 6 Message 1 Message 2 Message 3 Message 4 Message 5 Message 6	Messages segmented and reconstructed in order
11	 Sender Receiver Message 1 Message 2 Message 3 Message 4 Message 5 Message 6 Message 1 Message 2 Message 5 Message 6 Message 3 Message 4	Messages segmented and reconstructed out of order
12	 Sender Receiver Message 1 Message 2 Message 3 Message 4 Message 5 Message 6 Message 7 Message 8 Message 1 Message 2 Message 5 Message 6 Message 7 Message 8	Messages segmented and reconstructed partially lost
13	 Sender Receiver Message 1 Message 2 Message 3 Message 4 Message 5 Message 6 Message 7 Message 8 Message 1 Message 2 Message 7 Message 8 Message 5 Message 6	Messages segmented and reconstructed out of order and partially lost

Chapter 5

Channel Categorization and Matching

In this section, I investigated 4 different channel types: Named pipes, MQMS, TCP/UDP socket and HTTP channels, all of which are the most fundamental ones in Windows communication framework. By matching these channels to the communication model I verified generality of the modeling.

5.1 Named Pipes Channel

A named pipe is a named, one-way or duplex pipe for communication between the pipe server and one or more pipe clients. Both the server and client can read or write into the pipe. The pipe server and client can be on the same or different computers. In here we only consider one server V.S one client dual-trace. One server to multiple clients scenario can always be divided into multiple server/client dual-traces. We call the end of the named pipe instance. An instance can be a server instance or a client instance. All instances of a named pipe share the same pipe name, but each instance has its own buffers and handles, and provides a separate conduit for client/server communication.

A named pipe server responsible for the creation of the pipe, while clients of the pipe can connect to the server after it created. The creation and connection of a named pipe will return the handle ID of that pipe. As we mention before, each instance has its own handles, so the returned handle IDs of the pipe creation function of the server and pipe connection function from each client are different. These handler IDs will be used later on when messages are being sent or received to specify a pipe to which send.

5.1.1 Important Channel Parameters

There are many options for a named pipe. Some of them are critical in the perspective of the channel operations. In this sections we list the important ones.

Blocking/Non-Blocking

The named pipe channel can be opened in blocking mode or non-blocking mode. In blocking mode, when the pipe handle is specified in the ReadFile, WriteFile, or ConnectNamedPipe function, the operations are not completed until there is data to read, all data is written, or a client is connected. In non-blocking mode, ReadFile, WriteFile, and ConnectNamedPipe always return immediately. The operation fail if the channel is not ready for read, write or connection. However, since in this work we only aimed at locating the successful communication, as long as the operations success, we can locate them no matter the pipe is in blocking or non-blocking mode.

Synchronous/Asynchronous

Another critical option for named pipe channel is Synchronous/Asynchronous. In Synchronous mode, the ReadFile, WriteFile TracnsactNamedPipe and connectNamedPipe functions does not return until the operation it is performing is completed. That means we can retrieve the sent/receive message in the trace when the function return. When the channel is enable overlapped mode, the ReadFile, WriteFile TracnsactNamedPipe and connectNamedPipe functions perform asynchronously. In the asynchronous mode, ReadFile, WriteFile, TransactNamedPipe, and ConnectNamedPipe operations return immediately regardless if the operations are completed. And if the function call return ERROR_IO_PENDING, the calling thread then call the GetOverlappedResult function to determine the results. For the read operation, the message will be stored in the buffer indicated in the ReadFile function call when the read operation complete successfully.

5.1.2 Send/Receive Scenarios

In section **??**, I defined 13 scenarios of the general communication channel. In this section, I check their existence for named pipe channel. Some important properties of named pipe channel, which affecting the happening of the scenarios are listed in Table 5.1. The result of the existence of the scenarios is list in Table5.2 with the explanation by the affecting properties.

Table 5.1: Named Pipe Channel properties

Property	**Description**
1	All message going into the pipe will go out in order
2	The receiver can read multiple times to get the whole message, when he send message size is larger than the receiver's buffer
3	The receiver can only read contents from one write operation on the other end of the pipe in its one read operation
4	Read/Write operation return immediately when error occurs

Table 5.2: Send/Receive Scenarios of Named Pipe

Scenario	**Existence**	**Comment**
Scenario 1	YES	Property 1
Scenario 2	NO	Property 1
Scenario 3	YES	Property 4
Scenario 4	YES	Property 2
Scenario 5	NO	Property 1
Scenario 6	NO	Property 1
Scenario 7	NO	Property 1
Scenario 8	NO	Property 3
Scenario 9	NO	Property 3
Scenario 10	NO	Property 3
Scenario 11	NO	Property 3
Scenario 12	NO	Property 3
Scenario 13	NO	Property 3

5.1.3 Function Calls In Each Communication Stage

As we talk in the parameters section, Synchrounous and Asychronous mode affect the functions used to complete the send and receive operation as well as the operation of the functions. In the follow subsections, we will list the related functions for the named pipe channel for both synchronous mode and asynchronous mode. The create channel functions for both modes are the same but with different input parameters. The functions for send and receive message are also the same for both case. However, the operation of the send and receive functions are different for different mode. In addition, extra functions are being called to check the status of message sending or receiving in asynchronous mode.

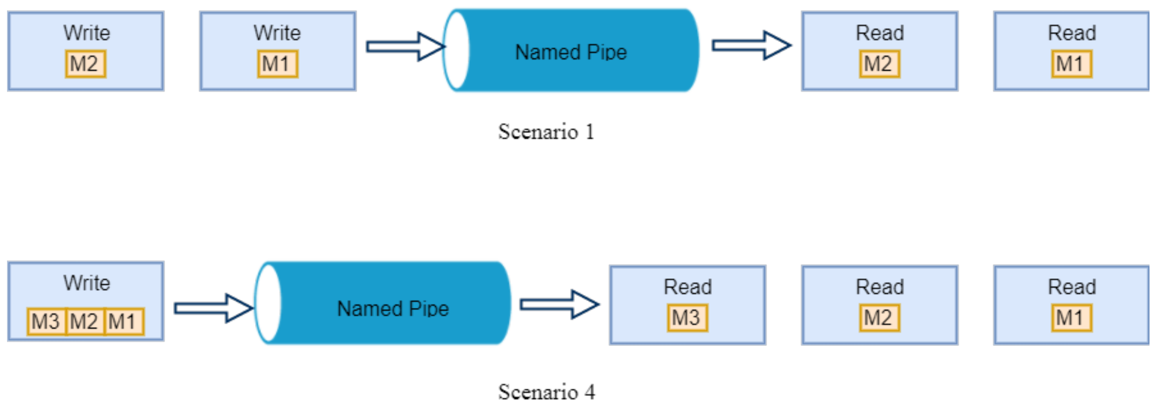


Figure 5.1: Two successful write/read operation scenarios. Each blue block indicate a single read or write operation.

Synchronous

We list all the functions that needed to locate an messaging event in a dual-trace in Table5.3 for synchronous named pipe. The Channel Open Functions indicate how the channel being opened/created in server and client sides, and they are different. The file name is an input parameter for CreateNamedPipe and CreateFile function. The client and server of the same pipe use the same file name. This is an important parameter to identify the pipe between the server and client in the traces. The File name is stored in the RCX register when the function is being called. The file handle is a integer return value from the CreateNamedPipe and CreateFile functions call. It will be stored in RAX register when the function return. This handle will be used as the identifier of a pipe in the client or server later on. The handle is different for server and each client even they connected to the same pipe. The send or receive message functions are the same in server and client. the file handle generated when the channel created are stored in register RCX when the WriteFile and ReadFile functions are being called. RDX holds the address of the buffer for message send or receive. The actual size of the message being sent or received are store in R9 when the function return.

Asynchronous

The functions used in Asynchronous mode for create channel, send and receive message are the same as those used in synchronous mode. However, the ReadFile and WriteFile functions run asynchronously when the channel is asynchronous. This means the function will return immediately, even if the operation has not been completed. If the operation is complete when the function

Table 5.3: Functions for communication stages definition of synchronous named pipe

Stage	**Server**		**Client**	
	Function	**Parameters**	**Function**	**Parameters**
Channel Open	CreateNamed-Pipe	RAX: File Handler	CreateFile	RAX: File Handler
		RCX: File Name		RCX: File Name
Message Send/Re-ceive	WriteFile	RCX: File Handle	WriteFile	RCX: File Handle
		RDX: Buffer Address		RDX: Buffer Address
		R9: Message Length		R9: Message Length
	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

returns, the return value indicates the success or failure of the operation. Otherwise the functions return zero and GetLastError returns ERROR_IO_PENDING. In this case, the calling thread must wait until the operation has finished. The calling thread must then call the GetOverlappedResult function to determine the results. This means besides looking for ReadFile and WriteFile function calls in the traces, the GetOverlappedResult function should be checked in the traces to get the full result of the ReadFile or WriteFile operations. There are two scenarios of the successful communication in the dual-trace for asynchronous mode. The first one is exactly the same as synchronous situation. The second one involves the functions listed in Table5.4. In the later one, Read operation searching is bit more complicated, since ReadFile function has to be located first to get the buffer address, and then GetOverlappedResult function return should be search for the message retrieval from the memory.

5.2 MQMS Channel

Message Queuing (MSMQ) is designed for communication between applications which is running at different times across heterogeneous networks and systems that may be temporarily offline. 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. [6]

However, in this work only one sending application versus one receiving application case is considered. Multiple sender to multiple receiver scenario can always be divided into multiple sender/receiver dual-traces.

The sending or receiving application can create the queue or use the existing one. However,

Table 5.4: Functions for additional communication type definition of asynchronous named pipe

Stage	Server		Client	
	Function	**Parameters**	**Function**	**Parameters**
Channel Open	CreateNamed-Pipe	RAX: File Handler	CreateFile	RAX: File Handle
		RCX: File Name		RCX: File Name
Message Send/Re-ceive	WriteFile	RCX: File Handle	WriteFile	RCX: File Handle
		RDX: Buffer Address		RDX: Buffer Address
		R9: Message Length		R9: Message Length
	ReadFile	RAX: File Handle	ReadFile	RCX: File Handle
		RDX: Buffer Address		RDX: Buffer Address
		R9: Message Length		R9: Message Length
	GetOver-lappedResult	RCX: File Handler	GetOver-lappedResult	RCX: File Handler
		RDX: OVERLAPPED		RDX: OVERLAPPED
Channel Close	CloseHandle	RCX: File Handler	CloseHandle	RCX: File Handler

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

5.2.1 Important Channel Parameters

Same as Named pipe channels, MSMQ channel have many options or settings affecting the operations happen in the channels. In this section I listed those ones affecting the dual trace analysis.

Synchronous/Asynchronous Receiving

In Message Queuing channels, only receiving can operate in asynchronous mode. When synchronously reading messages, the input parameters fnReceiveCallback and lpOverlapped are set to NULL. The calling thread is blocked until a suitable message is available or a time-out occurs for the receiving function. When asynchronously reading messages, MQReceiveMessage returns MQ_OK if a suitable message is found. Otherwise, MQReceiveMessage returns immediately with the return value MQ_INFORMATION_OPERATION_PENDING. This return value indicates that the operation is pending and will be completed as soon as a suitable message can be found. Further operations needed to get the message later on. More details about the further operations will be described in Section 5.2.1

Asynchronous Receiving With Call Back Functio or Completion port

When the receiving operates in asynchronous mode, completion ports or call back function can be used for the asynchronously reading.

In the completion port using situation, MQGetOverlappedResult is called to retrieve the success or error code from the OVERLAPPED structure. If no message is received before the time-out period elapses, an error is returned, and the start routine returns, terminating the thread in an implicit call to ExitThread.

In the call back function using situation, a call back function's pointer is given when the receiving function is being called. The call back function will perform its task as long a message was received or the time-out interval supplied by the caller elapsed.

Message Properties Description Structure

5.2.2 Send/Receive Scenarios

In this section, I check the scenarios existence for message queuing. Some important properties of MSMQ channel, which affecting the happening of the scenarios are listed in Table 5.5. The result of the existence of the scenarios is list in Table5.6 with the explanation by the affecting properties.

Table 5.5: Named Pipe Channel properties

Property	Description
1	All message going into the pipe will go out in order
2	The receiver can read multiple times to get the whole message, when he send message size is larger than the receiver's buffer
3	The receiver can only read contents from one write operation on the other end of the pipe in its one read operation
4	Read/Write operation return immediately when error occurs

5.2.3 Function Calls In Each Communication Stage

We list all the functions that needed to locate an messaging event in a dual-trace in Table5.3 for MSMQ. The Channel Open Functions indicate how the channel being opened in sending and receiving sides. The Queue Format Name is an input parameter for MQOpenQueue function. This is an important parameter to identify the queue. The File name is stored in the RCX register when the function is being called. The queue handle is a integer return value from the MQOpenQueue functions call. It will be stored in RAX register when the function return. The handle is different

Table 5.6: Send/Receive Scenarios of Named Pipe

Scenario	**Existence**	**Comment**
Scenario 1	YES	Property 1
Scenario 2	NO	Property 1
Scenario 3	YES	Property 4
Scenario 4	YES	Property 2
Scenario 5	NO	Property 1
Scenario 6	NO	Property 1
Scenario 7	NO	Property 1
Scenario 8	NO	Property 3
Scenario 9	NO	Property 3
Scenario 10	NO	Property 3
Scenario 11	NO	Property 3
Scenario 12	NO	Property 3
Scenario 13	NO	Property 3

for each application using the same queue. The queue handle generated when the queue is opened by an application are stored in register RCX when the MQSendMessage or MQReceiveMessage functions are being called for message sending and receiving. RDX holds the address of the sending message structure for message sending while R9 holds the address of the receiving message structure for message receiving. For message receiving, the operation can be synchronous or asynchronous. For the asynchronous receiving, an callback function's pointer is indicated in the parameter fnReceiveCallback which is pushed on the stack if the callback function is employed. If the call back function is not employed, MQGetOverlappedResult should be called to get the received message.

Synchronous Receiving

The functions used for defining a communication channel and events are listed in Table5.7.

Asynchronous Receiving with Call Back Functions

The functions used for defining a communication channel and events are listed in Table5.8.

Asynchronous Receiving without Call Back Functions

The functions used for defining a communication channel and events are listed in Table5.9.

Table 5.7: Functions for communication stages of MSMQ for synchronous receiving

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

Table 5.8: Functions for communication stages of MSMQ for asynchronous receiving with call back function

Stage	**Function**	**Parameters**
Channel Open	MQOpenQueue	RAX: Queue Handler
		RCX: Queue Format Name
Message Send/Receive	MQSendMessage	RCX: Queue Handle
		RDX: Message description structure Address
	MQReceive-Message	RCX: Queue Handle
		R9: Message description structure Address
		Stack(The Third One): Pointer of the call back function
Channel Close	MQCloseQueue	RCX: Queue Handler

5.3 TCP/UDP Socket Channel

5.4 HTTP Channel

Table 5.9: Functions for communication stages of MSMQ for asynchronous receiving without call back function

Stage	Function	Parameters
Channel Open	MQOpenQueue	RAX: Queue Handler
		RCX: Queue Format Name
Message Send/Receive	MQSendMessage	RCX: Queue Handle
		RDX: Message description structure Address
	MQReceive-Message	RCX: Queue Handle
		R9: Message description structure Address
		Stack(The Fourth One): Overlap Structure address
	MQGetOver-lappedResult	RCX: Overlap Structure address
Channel Close	MQCloseQueue	RCX: Queue Handler

Chapter 6

Channel Rebuild Algorithms

6.1 Channel Rebuild Algorithm for TCP

Algorithm 1: Data struct for TCP Channel Rebuild Algorithm

```
struct {
    SocketSR trace1
    SocketSR trace2
} RebuiltChannel
struct {
    Socket            socket
    String            sendBytes
    String            recvBytes
    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 2: TCP Channel Rebuild Algorithm

```
Input: Trace1 and Trace2 from both sides of dual-trace
Output: All rebuilt TCP channels
$channels \leftarrow List\langle RebuiltChannel\rangle$
$trace1sockets \leftarrow searchSockets\,(trace1)$
$trace2sockets \leftarrow searchSockets\,(trace2)$
for $s1 \in trace1sockets$ do
    for $s2 \in trace2sockets$ do
        if $s1.local = s2.remote$ AND $s2.local = s1.remote$ then
            $channel.trace1Socket \leftarrow s1$
            $channel.trace2Socket \leftarrow s2$
            $channels.add\,(channel)$
$rebuildSendAndRecv\,(trace1, channels, True)$
$rebuildSendAndRecv\,(trace2, channels, Flase)$
```

Algorithm 3: searchSockets() Function for TCP Channel Rebuild 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 4: rebuildSendAndRecv() Function for TCP Channel Rebuild Algorithm

```
Function rebuildSendAndRecv($trace, channels, isTrace1$):
    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
                $send.lineNum \leftarrow currentline$
                $sr.sends.add(send)$
                $bytes \leftarrow$ data that sent by the function call
                $sr.sendBytes.append(bytes)$
        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
                $receive.lineNum \leftarrow currentline$
                $sr.receives.add(send)$
                $bytes \leftarrow$ data that received when the function return
                $sr.recvBytes.append(bytes)$
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 $sr$
```

6.2 Channel Rebuild Algorithm for UDP

Algorithm 5: Data struct for UDP Channel Rebuild Algorithm

```
Input: Trace1 and Trace2 from both sides of dual-trace
Output: All rebuilt UDP channels
struct {
    Socket trace1
    Socket trace2
    List⟨ SRPair⟩ trace1Totrace2
    List⟨ SRPair⟩ trace2Totrace1
} RebuiltChannel
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;
} Function
```

Algorithm 6: UDP Channel Rebuild Algorithm

```
Input: Trace1 and Trace2 from both sides of dual-trace
Output: All rebuilt UDP channels
$channels \leftarrow List\langle RebuiltChannel\rangle$
$trace1sockets \leftarrow searchSockets\,(trace1)$
$trace2sockets \leftarrow searchSockets\,(trace2)$
for $s1 \in trace1sockets$ do
    for $s2 \in trace2sockets$ do
        if $s1.local = s2.remote$ AND $s2.local = s1.remote$ then
            $channel.trace1Socket \leftarrow s1$
            $channel.trace2Socket \leftarrow s2$
            $channels.add\,(channel)$
$rebuildSendAndRecv\,(trace1, trace2, channels)$
```

Algorithm 7: rebuildSendAndRecv() Function for UDP Channel Rebuild Algorithm

```
Function rebuildSendAndRecv($(trace1, trace2, channels)$):
    $trace1Sends, trace2Sends, trace1Receives, trace2Receives \leftarrow List\langle Function\rangle$
    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
        $SRPair.send \leftarrow s$ for $r \in trace2Receives$ do
            if $s.bytes = r.bytes$ AND $s.bytes = r.bytes$ then
                $SRPair.recv \leftarrow r$
            $channels$
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
        $func.hanle \leftarrow handle\ list.add(func)$
```

6.3 Channel Rebuild Algorithm for Named Pipes

Algorithm 8: Data struct for Named Pipe Channel Rebuild Algorithm

```
struct {
    SocketSR trace1
    SocketSR trace2
} RebuiltChannel
struct {
    Socket             socket
    String             sendBytes
    String             recvBytes
    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 9: Named Pipe Channel Rebuild Algorithm

```
Input: Trace1 and Trace2 from both sides of dual-trace
Output: All rebuilt TCP channels
$channels \leftarrow List\langle RebuiltChannel\rangle$
$trace1sockets \leftarrow searchSockets\,(trace1)$
$trace2sockets \leftarrow searchSockets\,(trace2)$
for $s1 \in trace1sockets$ do
    for $s2 \in trace2sockets$ do
        if $s1.local = s2.remote$ AND $s2.local = s1.remote$ then
            $channel.trace1Socket \leftarrow s1$
            $channel.trace2Socket \leftarrow s2$
            $channels.add\,(channel)$
$rebuildSendAndRecv\,(trace1, channels, True)$
$rebuildSendAndRecv\,(trace2, channels, Flase)$
```

Algorithm 10: searchSockets() Function for Named Pipe Channel Rebuild 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 11: rebuildSendAndRecv() Function for Named Pipe Channel Rebuild Algorithm

```
Function rebuildSendAndRecv($trace, channels, isTrace1$):
    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
                $send.lineNum \leftarrow currentline$
                $sr.sends.add(send)$
                $bytes \leftarrow$ data that sent by the function call
                $sr.sendBytes.append(bytes)$
        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
                $receive.lineNum \leftarrow currentline$
                $sr.receives.add(send)$
                $bytes \leftarrow$ data that received when the function return
                $sr.recvBytes.append(bytes)$
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 $sr$
```

Chapter 7

Prototype

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.

7.0.1 User Defined Communication Type

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.

Function Calls in the Traces

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 Figure7.1 there is an action item ”Add to Communication type” in the right click menu of the function entry. Figure 7.2 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.

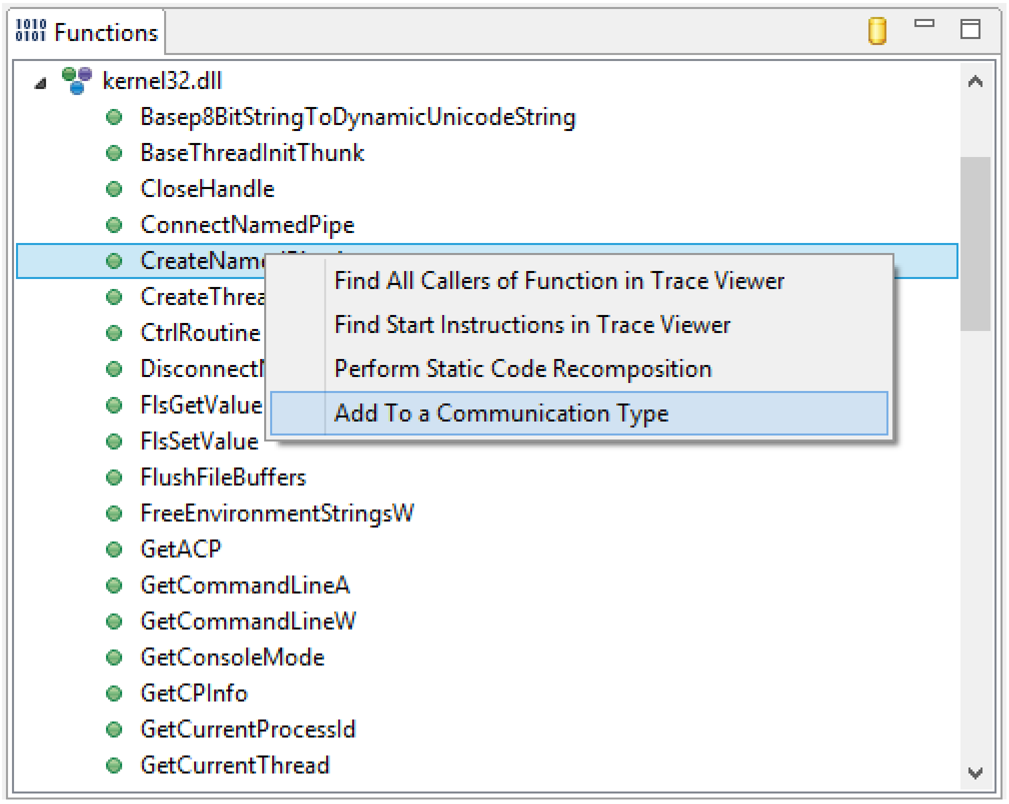
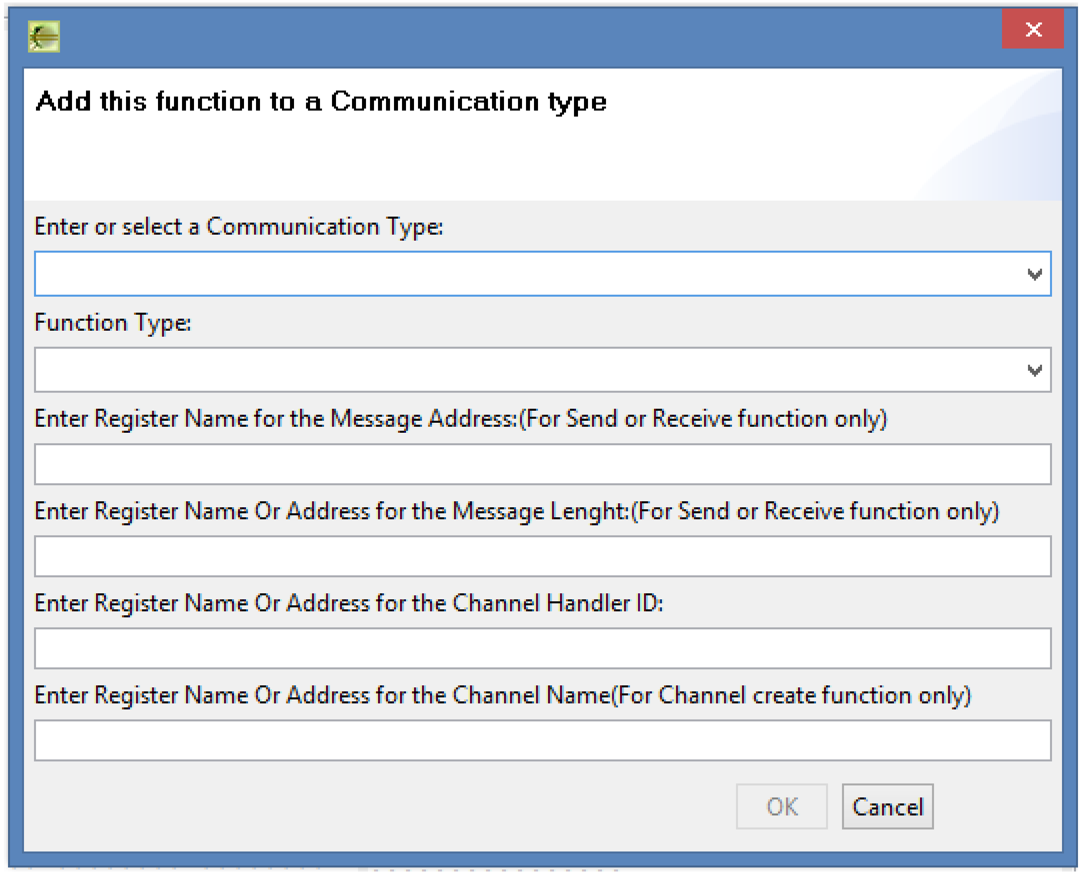


Figure 7.1: Add function to a Communication type from Functions View


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 7.2: Dialog to input information for a function adding to a communication type

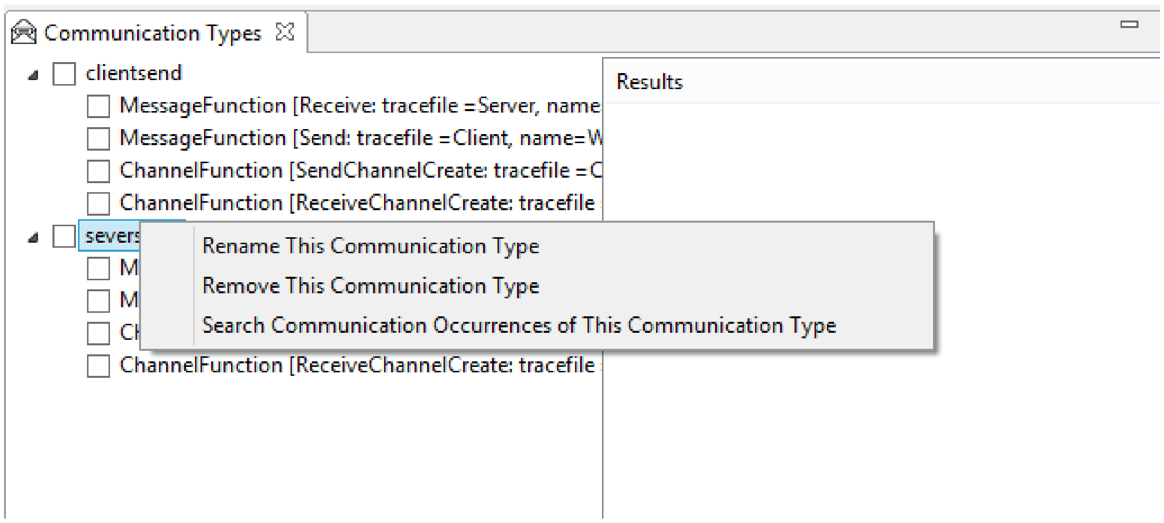
Communication Type Data Structure

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>
```

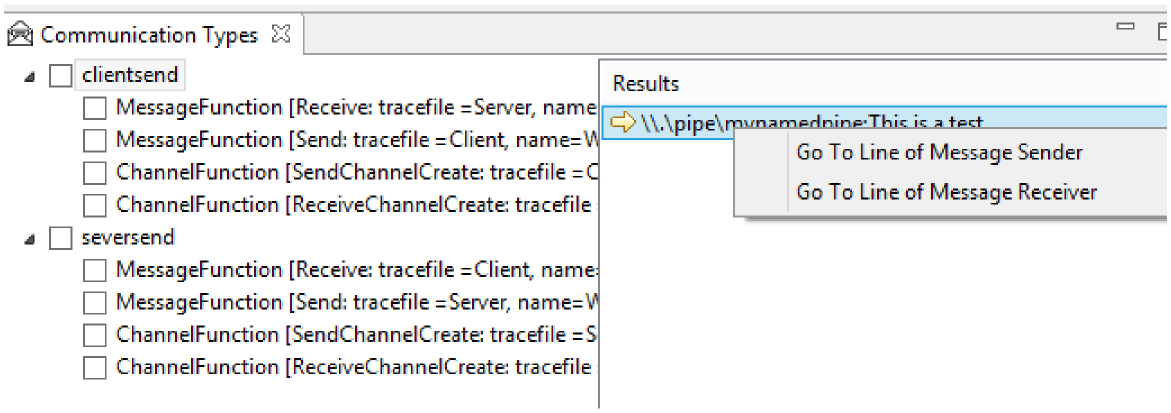
Communication Type View

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 7.3. 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 Figure7.4. Message content in the memory view will be shown as well.


Communication Types
Results
Rename This Communication Type
Remove This Communication Type
Search Communication Occurrences of This Communication Type

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

Figure 7.3: New View: Communication Type View



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

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

Figure 7.4: Right Click menu to navigate to send and receive event in the traces

7.0.2 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.

7.0.3 Matching Event Visualization and Navigation

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 8

verification

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.

8.0.1 Test and 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.

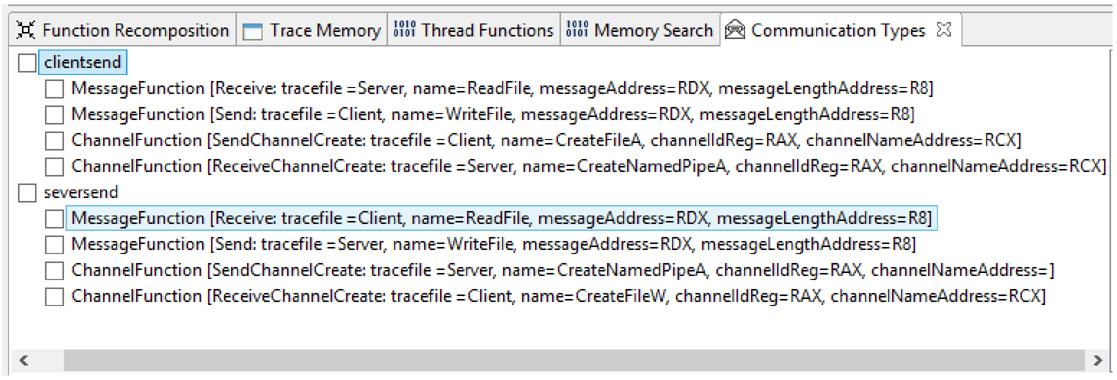
In each test case, there are three test steps: 1. define the communication type by adding channel creating functions and message send/receive functions of server and client sides. 2. search for the events of the defined communication type. 3. for the occurrence of the events, navigate to the trace instruction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

8.0.2 Result

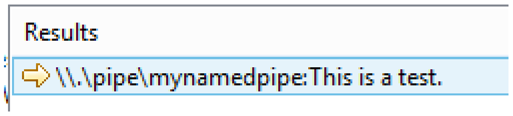
We used the dual-trace provided by DRDC and follow the experiment and verification design to conduct this test. Figure8.1 shows that the user defined clientsend and serversend communication types are shown in the communication type view as well as the functions consist of the communication types. Figure8.3 shows the search result of clientsend communication type, while Figure8.3 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. Figure8.4 shows the server was sending out a message: This is an answer.



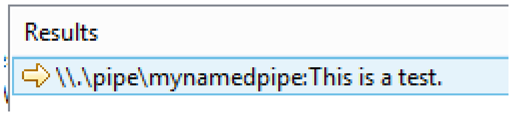
```
☐ clientsend
    ☐ MessageFunction [Receive: tracefile =Server, name=ReadFile, messageAddress=RDX, messageLengthAddress=R8]
    ☐ MessageFunction [Send: tracefile =Client, name=WriteFile, messageAddress=RDX, messageLengthAddress=R8]
    ☐ ChannelFunction [SendChannelCreate: tracefile =Client, name=CreateFileA, 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 8.1: Defined clientsend and serversend communication types in Communication View



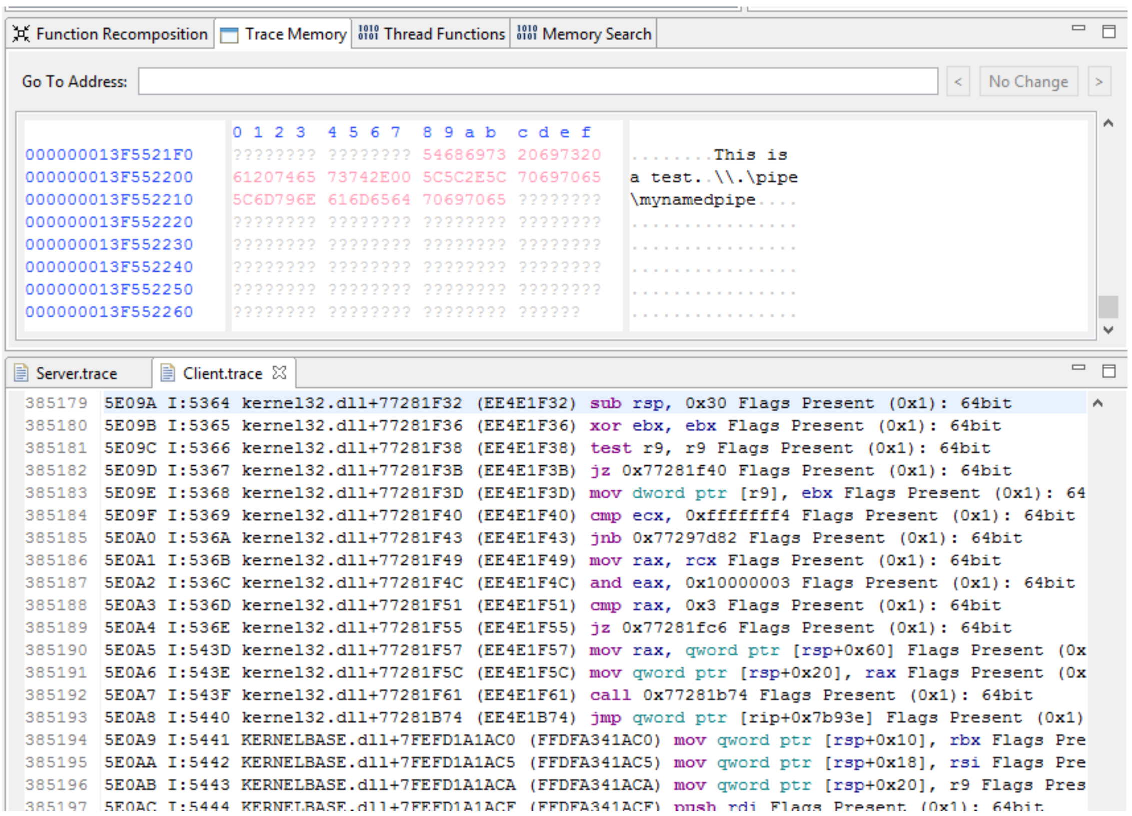
```
Results
\\.\pipe\mynamedpipe:This is a test.
```

Figure 8.2: the search result of clientsend communication type



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

Figure 8.3: 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 8.4: instruction view and memory view updated correctly

Chapter 9

Conclusions

9.1 Limitations

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

9.1.1 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.

9.1.2 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.

9.1.3 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.

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

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

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

[7] 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.