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

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

# Modeling

I investigated some common used communication methods divide the communication methods into two categories based on their data transmission properties. Based on this investigation, I modelled the communication of two programs. I also the dual-trace of two communicating programs in the perspective of communication analysis. These two models are the foundation to decide how communications being identified from the dual-trace and how to present them to the user.

## 1.1 Communication Categorization and Communication Methods

The goal of this work is to identify the communications from the dual-trace. We need to understand the properties of the communications to identify them. In general, there are two types of communication: reliable and unreliable in the perspective of their reliability of data transmission. The reason to divide the communication methods into these two categories is that the data transmission properties affect the mechanism of the identification of the communications fall in different categories. In the following two subsections, I summarize the characteristics of these two communication categories. The communication methods list in Table1.1 will be discussed further to provide more concrete comprehension.

Table 1.1: Communication Methods Discussed in This Work

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

### 1.1.1 Reliable Communication

A reliable communication guarantees the data being sent by one endpoint of the channel always received losslessly 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. For some communication methods, a channel can be closed without waiting the completion of all data transmission. In this case, the receive data union can be a sub string of the send data union.

### 1.1.2 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.

### 1.1.3 Communication Methods

In this section, I describe the mechanism and the basic data transfer characteristics of each communication method in Table1.1 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 Figure1.1.

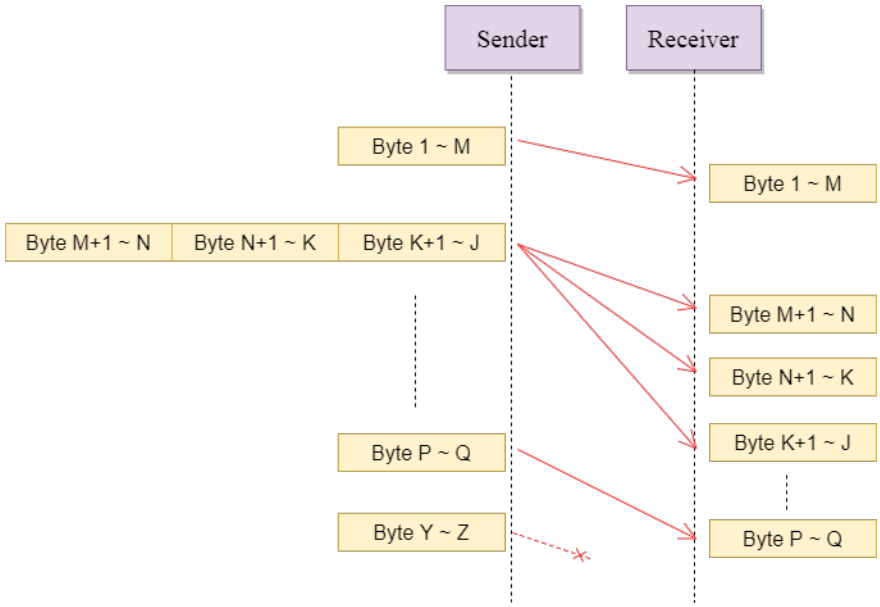


Figure 1.1: Data Transfer Scenarios for Named Pipe

**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 Figure1.2.

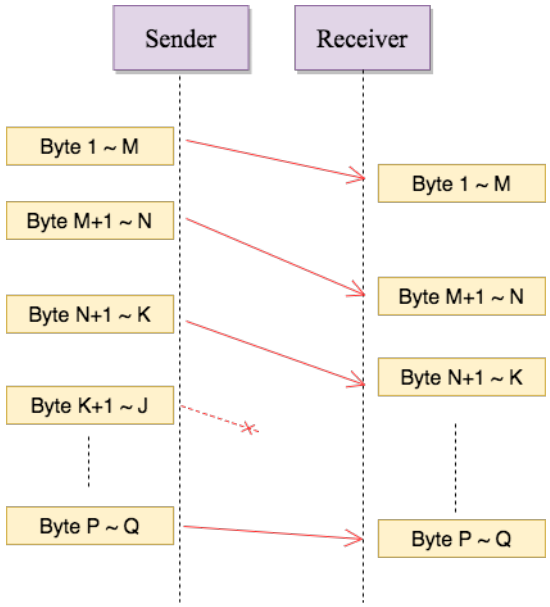


Figure 1.2: Data Transfer Scenarios for Message Queue

**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 Figure1.3.

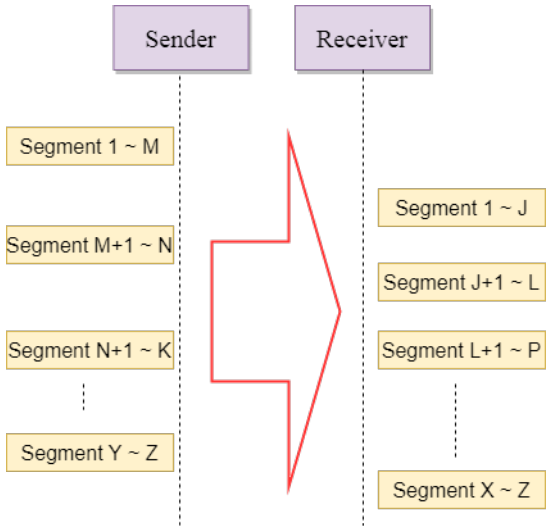


Figure 1.3: Data Transfer Scenarios for TCP

**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 Figure1.4.

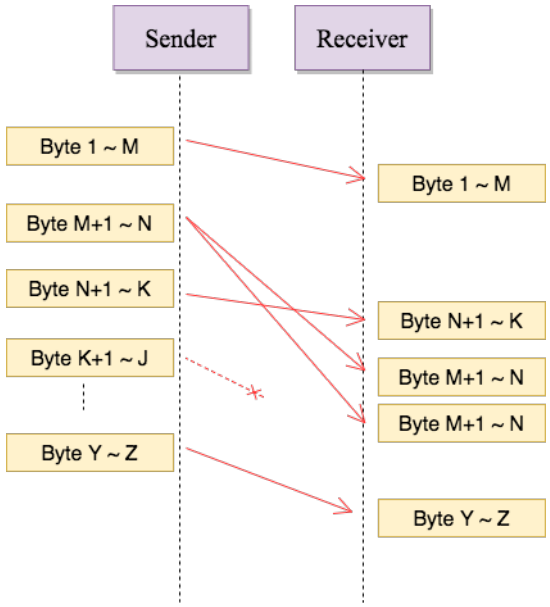


Figure 1.4: Data Transfer Scenarios for UDP

## 1.2 Model of Communication

This section define the communication of two programs. 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 re-opened 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. There are many communication methods for the data transfer communications, but all of them are compatible to this communication definition.

A communication $Co$ is defined by the 2-tuple $\langle ep, c\rangle$, where $ep$ is a set $\{e_x : x = 0, 1\}$ for the two endpoints communicating with each other though the channel $c$. The endpoint $e_x$ is defined by the 3-tuple $\langle h_x, ds_x, dr_x\rangle$. $h_x$ is the handle created within a process for subsequent data transfer operations. $ds_x$ is the sequence of packets sent in the sending operations of $h_x$ while $dr_x$ is the sequence of packets received in the receiving operations of $h_x$. $e_0$ is created in process $p$ and $e_1$ is created in process $q$. Let $ds_x = (ps_{x,i} : 0 \leqslant i \leqslant M_x)$ and $dr_x = (pr_{x,j} : 0 \leqslant j \leqslant N_x)$ in which $ps_{x,i} = \langle ts_{x,i}, ss_{x,i}\rangle$ and $pr_{x,i} = \langle tr_{x,j}, sr_{x,j}\rangle$. $ts_{x,i}$ and $tr_{x,j}$ are the logical time when the packet being sent and received. $ss_{x,i}$ and $sr_{x,j}$ are the string payloads being sent and received. The string

payloads can treated as sequence in the same order of the packets, $pls_x = (ss_{x,i} : 0 \leqslant i \leqslant M_x)$ and $plr_x = (sr_{x,j} : 0 \leqslant j \leqslant N_x)$. $\forall ps_{x,i} \in ds_x$, $ts_{x,k} \leqslant tr_{x,l}$ if $k \leqslant l$; $\forall pr_{x,i} \in dr_x$, $tr_{x,k} \leqslant tr_{x,l}$ if $k \leqslant l$;

There are two sets of preservation of this definition. One set is for the reliable communication while the other is for the unreliable one. There are content preservation and timing preservation in each preservation set.

**Preservation for reliable communication:**

- *Content Preservation:* Let $S_x$ be the concatenation of $\forall ss_{x,i} \in pls_x$ and $R_x$ be the concatenation of $\forall sr_{x,i} \in plr_x$. Then, $R_0$ is a sub string of $S_1$ and $R_1$ is a sub string of $S_0$.

- *Timing Preservation:* Let $S_{x,k}$ be the concatenation of $\forall ss_{x,i} \in pls_x, 0 \leqslant k \leqslant M_x$ and $R_{x,l}$ be the concatenation of $\forall sr_{x,i} \in plr_x, 0 \leqslant l \leqslant N_x$. If $S_{0,k}$ is $R_{1,l}$, then $ts_{0,k} \leqslant tr_{1,l}$. If $S_{1,k}$ is $R_{0,l}$, then $ts_{1,k} \leqslant tr_{0,l}$.

**Preservation for unreliable communication:**
$\forall sr_{0,j} \in plr_0, \exists ss_{1,i} \in pls_1$ and $\forall sr_{1,j} \in plr_1, \exists ss_{0,i} \in pls_0$ such that

- *Content Preservation:* $sr_{0,j} = ss_{1,i}$ and $sr_{1,j} = ss_{0,i}$

- *Timing Preservation:* $tr_{0,j} > ts_{1,i}$ and $tr_{1,j} > ts_{0,i}$

The terminology of using in this definition can be found in 3.1.

In the following two examples, $h_0$ and $h_1$ are the handles for the two endpoints of the communication. $ds_0$, $dr_0$, $ds_1$ and $dr_1$ are the sequence of packets sent and received by the endpoints. The string payloads are listed in blue and red in the figures.

Figure1.5 is an example of the reliable communication. In this example, $ss_{0,0}$ = "$ab$"; $ss_{0,1}$ = "$cde$"; $ss_{0,2}$ = "$fgh$", $sr_{1,0}$ = "$abc$"; $sr_{1,1}$ = "$def$"; $ss_{1,2}$ = "$gh$" and on the other direction $ss_{1,0}$ = "$mno$"; $ss_{1,1}$ = "$pqr$"; $ss_{1,2}$ = "$stu$", $sr_{0,0}$ = "$mnop$"; $sr_{0,1}$ = "$qrstu$" . $ss_{0,0}.ss_{0,1}.ss_{0,2} = sr_{1,0}.sr_{1,1}.ss_{1,2}$ = "$abcdefgh$" and $ss_{1,0}.ss_{1,1}.ss_{1,2} = sr_{0,0}.sr_{0,1}$ = "$mnopqrstu$" satisfy the content preservation. The timing in this example are: $ts_{1,0} < ts1, 1 < tr_{0,0} < ts_{1,2} < tr_{0,1}$ and $ts_{0,0} < ts_{0,1} < tr_{1,0} < ts_{1,2} < tr1, 1 < tr_{1,2}$. The following statements of this example satisfy the timing preservation. $sr_{0,0}$ = "$mnop$" is the sub string of $ss_{1,0}.ss_{1,1}$ = "$mnopqr$", $sr_{0,0}.sr_{0,1}$ = "$mnopqrstu$" is the sub string of $ss_{1,0}.ss_{1,1}.ss_{1,2}$ = "$mnopqr$"$stu$, $sr_{1,0}$ = "$abc$" is the sub string of $ss_{0,0}.ss_{0,1}$ = "$abcde$", $sr_{1,0}.sr_{1,1}$ = "$abcdef$" and $sr_{1,0}.sr_{1,1}.sr_{1,2}$ = "$abcdefgh$" are the sub string of $ss_{0,0}.ss_{0,1}.ss_{0,2}$ = "$abcdefg$"

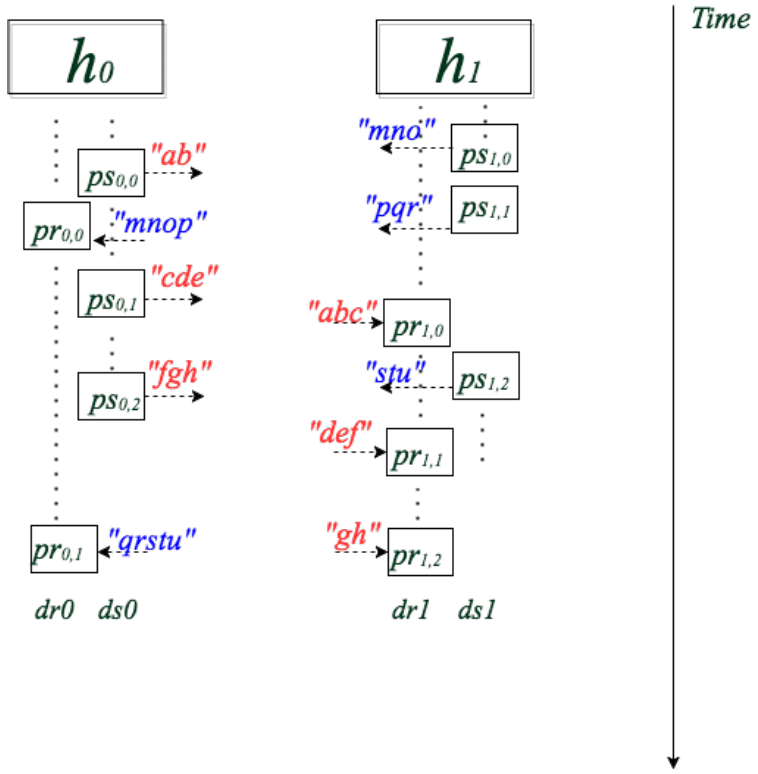


Figure 1.5: Example of Communication

Figure1.6 is an example of the unreliable communication. In this example, $sr_{1,0} = ss_{0,1} =$ "$cde$", $tr1,0 > ts_{0,1}$; $sr_{1,1} = ss_{0,2} =$ "$fi$",$tr1,1 > ts_{0,2}$; $sr_{0,0} = ss_{1,0} =$ "$gh$", $tr_{0,0} > ts_{1,0}$; $sr_{0,1} = ss_{1,1} =$ "$ijklm$", $tr_{0,1} > ts_{1,1}$; $sr_{0,2} = ss_{1,2} =$ "$n$", $tr_{0,2} > ts_{1,2}$. All of these satisfy the content preservation and timing preservation of the unreliable communication.

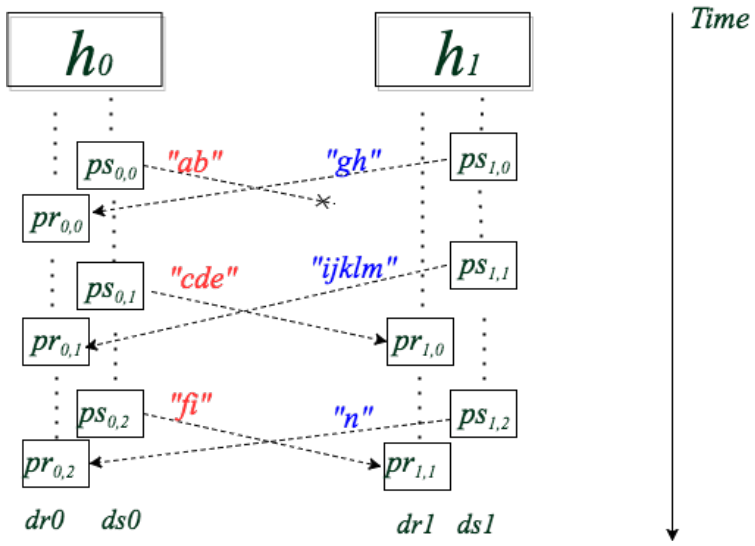


Figure 1.6: Example of Communication

## 1.3 Model of the Dual-Trace of Two Communicating Programs

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. In this section, I model the execution trace in the dual-trace. The communication identification is among the information of this model.

A dual-trace consist of two execution traces $\{trace1, trace2\}$ . An execution trace is defined as a sequence $trace = (line_k, 0 \leqslant l \leqslant K)$. $line_k$ in a $trace$ is a 3_tuple $\langle ins, mem, fi\rangle$ where $ins$ is the assembly instruction, $mem$ is memory changed by this instruction and $fi$ is function call information indicating if this is the function call or return. A function $eventfilter\,()$ is defined to generate the event level trace $event\_trace$ from the original $trace$. $event\_trace = eventfilter\,(trace, funcset)$, where $funcset = \{func_l, 0 \leqslant L \leqslant Q\}$ is a set of the concerned events' function information.Each concerned event's function information can be described a tuple $langle funN, type, pars\rangle$ where $funN$ is the concerned event's function name, $type$ can only be

one of these four event types: only be channel open, channel close, data send and data receive and $pars$, is the parameter information list. The output of this function $event\_trace$ is a sequence of events $(event_m, 0 \leqslant l \leqslant M)$. Only the concerned events in the $funcset$ are filtered in this sequence, all other information in the original trace are ignored. Each event in the trace corresponds to a system function call and is defined as a 4_tuple $\langle funN, startline, endline, type \rangle$. In this tuple $funN$ is the name of the called function, $startline$ is the line number where the function was being called, $endline$ is the line number where the function returned and $type$ is the event type. The events in the $event\_trace$ are interleaving events among multiple handles. Function $streamfilter\,()$ is defined to generate the stream level trace $event\_trace$ from the $event\_trace$. $stream\_trace = eventfilter\,(event\_trace)$. The output $stream\_trace$ is a set of stream $\{stream_n, 0 \leqslant n \leqslant N\}$ in which each stream correspond to a handle and consist of 4 sub streams. So that $stream_n$ is a set $\{open\_stream, send\_stream, receive\_stream, close\_stream\}$. Each of these sub streams consist of a sequence of $event$ of a certain handle of the corresponding event types.

## 1.4 Element matching from Trace Modelling and Communication Modelling

The goal of this work is to identify the communication from dual-trace. In the modelling, this can be abstracted as finding the elements of the communication model in the dual-trace model. The element matching can be summarized in TableḂy known this matching, algorithms can be developed to identify the communications in the dual-trace model. The developed algorithm will be discuss in next chapter.

# Chapter 2

# Communication Identification Algorithms

In the Modeling Chapter, I elaborate the definition of the communications and the strategy of the communication identification. There are several major steps in the identification strategy. In this chapter, I list developed the algorithms for the key steps in the identification strategy.

## 2.1 Communication Identification Process

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

## 2.2 Communication Methods' Implementation in Windows

Finally I discuss the implementation of the communication methods and draw the function lists of the event types of some concerned communication methods. The function list of a communication method is one of the inputs for the communication identification.

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.

### 2.2.1 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.

### 2.2.2 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. Table2.1 lists the functions of the events for synchronous mode while Table2.2 lists the functions of the events for the asynchronous mode for a Named pipe communication.

Table 2.1: Function List of events 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 2.2: Function List of events 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** | GetOverlapped-Result | RCX: File Handler | GetOverlapped-Result | 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. Figure2.1 shows the channel set up process for a Named Pipe communication.

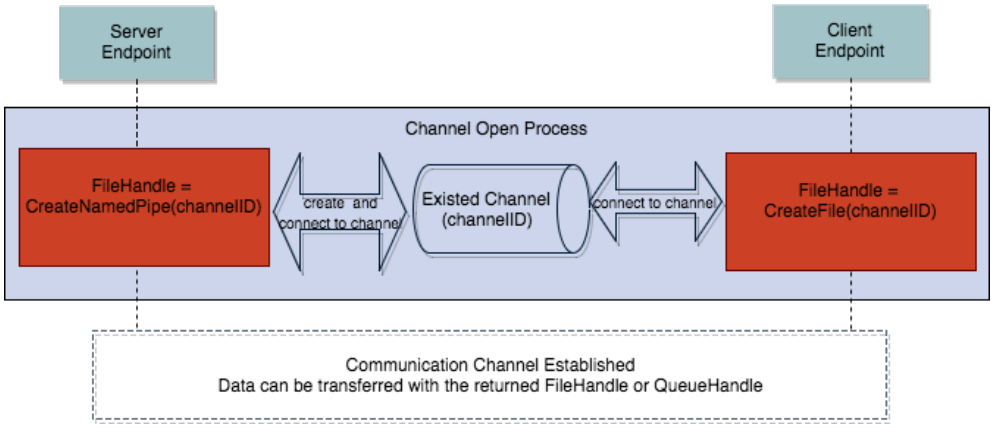


Figure 2.1: Channel Open Process for a Named Pipe

### 2.2.3 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. Table2.3 lists the functions for synchronous mode while Table2.4 and Table2.5 list the functions for the asynchronous mode with and without callback.

Table 2.3: Function List of events 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 2.4: Function List of events 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 2.5: Function List of events 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. Figure2.2 shows the channel set up process for a Message Queue communication.

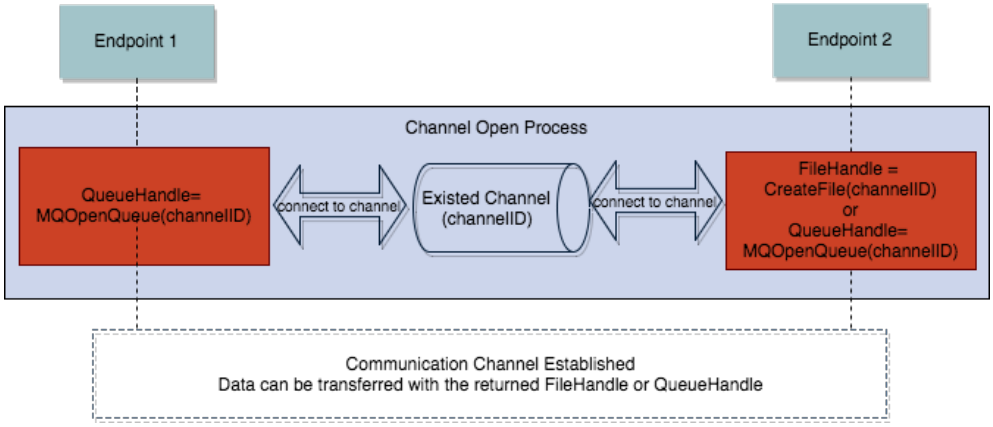


Figure 2.2: Channel Open Process for a Message Queue

### 2.2.4 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 2.6 lists the functions of a UDP or TCP communication.

Table 2.6: Function List of events 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. Figure2.3 shows the channel open process for TCP and UDP.

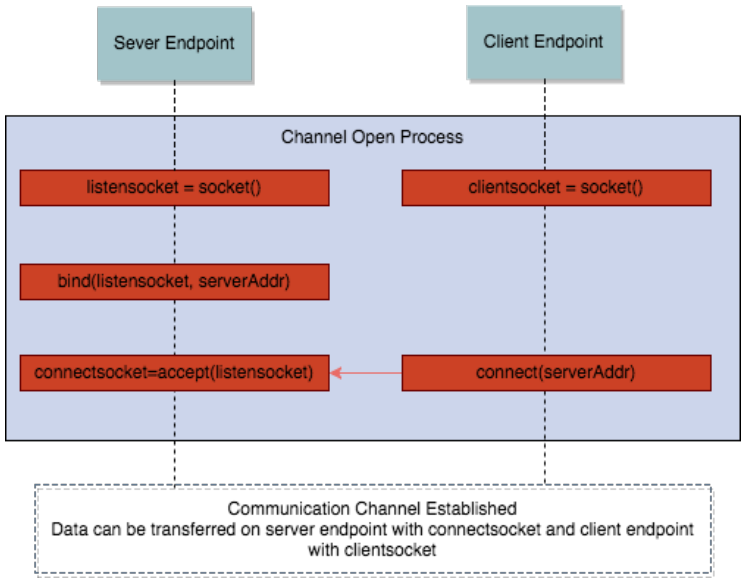


Figure 2.3: Channel Open Model for TCP and UDP

## 2.3 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.

### 2.3.1 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.

### 2.3.2 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 2.2 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.

## 2.4 Event Locating Algorithm

The concerned events in a communication are channel open, channel close, send and receive events. These events are identified as system function calls in this work. Each event can be completed by different function calls. For example, for the client endpoint in TCP communication method, both *socket* and *connect* function call are considered to be the channel open events. The functions list for a communication method is needed as a input of this algorithm. Tables in Section2.2 give the examples of function list of the events for some communication methods. The algorithm presented in this section is designed for locating all function calls provided in the function list as 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 function list usually contain a very small number of functions compared to the instruction line number in the execution trace, the time complexity of this algorithm is O(N+M) , N and M are the instruction line numbers of the two traces in the duel-trace.

**Algorithm 1: Event Locating Algorithm**

**Input:** dual-trace, function list

**Output:** two event lists for two separative traces in the dual-trace

```
1  eventLists ← Map⟨String, List⟨Event⟩⟩;
2  for trace ∈ dual-trace do
3      eventList ← List⟨Event⟩;
4      while not at end of trace do
5          for f ∈ functionList do
6              if Is function call of f then
7                  find function return instruction line;
8                  event.inputs ← get input parameters from this instruction line;
9                  event.outputs ← get output parameters and return value from the return
                    instruction line;
10                 event.type ← f.eventType;
11                 eventList.add (event);
12     eventLists.add (eventList);
13 return eventLists;
```

## 2.5 Endpoint Identification Algorithm

The events located in the two traces may correspond to different endpoints, the next step in the strategy 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 for one trace from the Event Locating Algorithm. So this algorithm should be run separately for both traces 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 identification for TCP and UDP server endpoints are slightly complicated than the other ones, due to its own channel open mechanism. 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.

**Algorithm 2: Endpoint Indentification Algorithm**

**Input:** event list of a trace

**Output:** endpoint list

```
1  endpoints ← Map⟨String, List⟨EndPoint⟩⟩;
2  for event ∈ eventList do
3      if event is a channel open event then
4          handle ← get the handle identifier from the function parameter list;
5          endpoint ← endpoints.get (handle);
6          if event is an accept (event) function call for TCP or UDP then
7              newHandle ← get the second socket handle identifier which is the return
                 value from the function parameter list;
8              endpoints.remove (handle);
9              endpoints.add (newHandle, endpoint);
10         if endpoint is null then
11             endpoint = New  EndPoint ();
12             endpoints.add (hanele, endpoint);
13         endpoint.openStream.add (event);
14     if event is a channel send event then
15         handle ← get the handle from the function parameter list;
16         endpoint ← endpoints.get (handle);
17         if endpoint is not null and endpoint.complete is False then
18             endpoint.sendStream.add (event);
19     if event is a channel receive event then
20         handle ← get the handle from the function parameter list;
21         endpoint ← endpoints.get (handle);
22         if endpoint is not null and endpoint.complete is False then
23             endpoint.receiveStream.add (event);
24     if event is a channel close event then
25         handle ← get the handle from the function parameter list;
26         endpoint ← endpoints.get (handle);
27         if endpoint is not null then
28             endpoint.closeStream.add (event);
29             endpoint ← True;
30 return endpoints;
```

## 2.6 Communication Identification Algorithm

The communication identification algorithm aims at identifying all the communication of a concerned communication method from the dual-trace. The input of this algorithm is the two endpoint lists for both traces in the dual-trace from the 'Endpoint Identification Algorithm'. The output of this algorithm is the communication list. Each communication recognized from the dual-trace contains two endpoints. The channel of a communication defined in Section1.2 is not explicitly represented in the output but it was implicitly used in this algorithm.

In the communication identification algorithm, it first try to match two endpoints to a channel only by their identifiers. In this level, the matching depends on channel open mechanisms which are different from communication method to communication method. For TCP and UDP the matching can be considered as local address and port of server endpoint matching with remote address and port of client endpoint. For Named Pipe, it uses the file name, while for Message Queue, it uses the queue name as the identifier for matching of two endpoints.

The first level matching can not guarantee the exact endpoints matching and channel identification. There are two situations which false positive error might emerge. Take Named Pipe for example, the first situation is multiple(more than two) interacting programs shared the same file or queue as their own channel. Even though the channels are distinct for each communication, but the file or queue used is the same one. For example, the Named Pipe server is connected by two clients using the same file. In the server trace, there are two endpoints found. In each client trace, there is one endpoint found. In the channel identification algorithm for the dual-trace of server and client1, there will be two possible identified channels, one is the real used one for server and client1 communication while the other is the false positive error actually is for server and client2. The endpoint in client1's trace will be matched by two endpoints in the server's trace. The second situation is the same channel is reused by the different endpoints in the same programs. For example, the Named Pipe server and client finished the first communication and then closed the channel. After a while they re-open the same file again for another communication. Since the first level matching is only base on the identifiers and the first and the second communications have the same identifier since they used the same file. Similar situations can also happen in Message Queue, TCP and UDP communication methods.

To reduce the false positive error, the second level matching should be applied, which is also being named as transmitted data verification algorithm. On top of the endpoint identifiers matching, further data verification should be applied to make sure the matching is reliable. This verification crossly compare the sent and received data of both endpoints in the first leve matching. If the transmitted data of the endpoints is considered to be identical, the matching is confirm, otherwise

it was a false positive error. However, we still can not exclude all the false positive errors, due to the data transmitted in two communication can be identical. Figure2.4 indicates the ineffective second level matching scenario and the effective one.

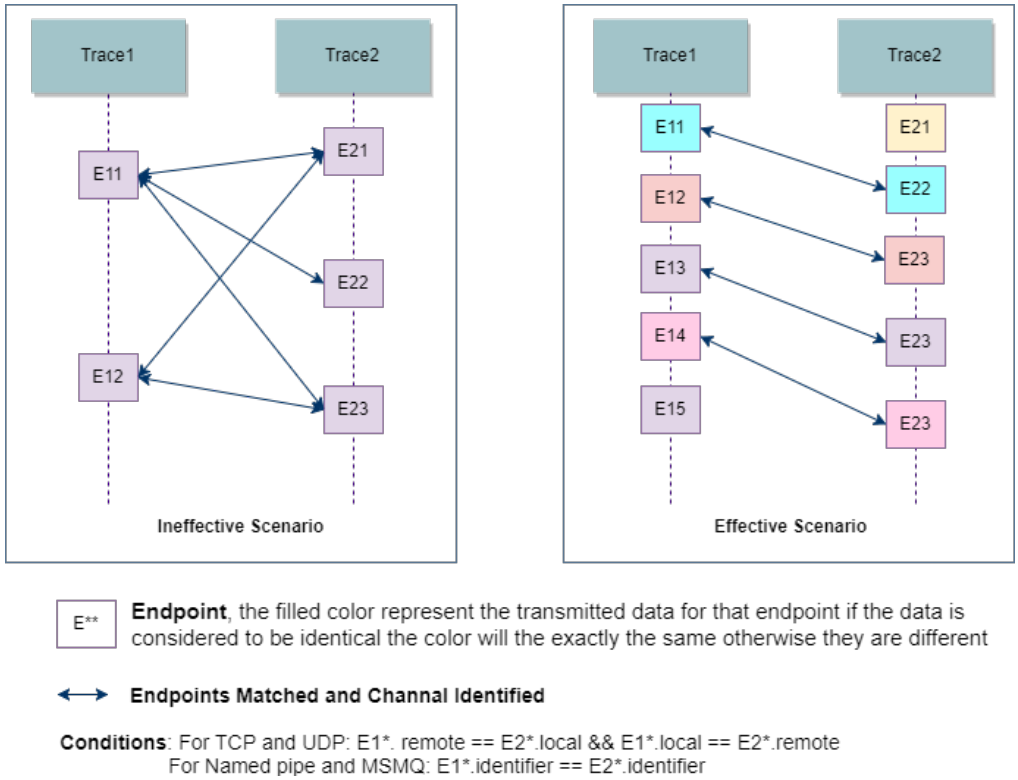


Figure 2.4: Second Level Matching Scenarios

The following subsections discuss the algorithms for these two level matching. In Section2.2, I elaborate the channel open process and the data transfer categories for the concerned communication methods. Based on the different channel process, two algorithms are developed for the communication identification, one is for Named Pipe and Message Queue, the other is for TCP and UDP. The inputs of the these two algorithms are the same, the two endpoint lists of the two traces of the analysing dual-trace. The output is the identified communications. Each communication contains the matched two endpoints from both sides of the traces and each endpoint contains its stream.

The data transfer characteristics divided the communication methods into reliable and unreliable transmissions. Named Pipe and TCP fall in the reliable category while Message Queue and UDP fall in the unreliable one. The second level matching algorithms are different for these two categories. The corresponding second level data matching algorithms are being used in the communication identification algorithms. The input of the transmitted data verification algorithms is

the streams of both endpoint while the output a boolean to indicate if the transmitted data of this two endpoint streams are matched and the verified data.

### 2.6.1 Communication Identification Algorithm for Named Pipe and Message Queue

For Named Pipe and Message Queue, only one channel open function is being called for each endpoint. So in the below algorithm, when it try to get the channel open event from the $endpoint.openStream$ list, only one event should be found and return. The channel identifier parameters can be found in the $event.inputs$ of the channel open event. The identifier for Named Pipe is the file name of the pipe while for Message Queue is the format queue name of the queue.

This algorithm finds out all the possible communications regardless some of them might be false positive errors. The called data verification algorithm is either "Matching by Data Streams Algorithm" for Named Pipe or "Matching by Data Events Algorithm" for Message Queue.

**Algorithm 3: Communication Indentification Algorithm for Named Pipe and Message Queue**

**Input:** two endpoint lists of the dual-trace

**Output:** communication list

```
communications ← Map⟨String, List⟨Communication⟩⟩;
for endpoint1 ∈ endpointList1 do
    openEvent1 ← get the event from endpoint1.openStream, which should only
     contain one event;
    channelId1 ← get the channel identifier from openEvent1.inputs;
    for endpoint2 ∈ endpointList2 do
        openEvent2 ← get the event from endpoint2.openStream, which should only
         contain one event;
        channelId2 ← get the channel identifier from openEvent2.inputs;
        if channelId1 == channelId2 then
            DataVerified ← Call the corresponding data verification algorithm. if
             DataVerified == True then
                communication = New Communication();
                communication.endpoint1 = endpoint1;
                communication.endpoint2 = endpoint2;
                communication.dataMatch ← The output from data verification
                 algorithm;
                communications.add (communication);
return channels;
```

### 2.6.2 Communication Identification Algorithm for TCP and UDP

For TCP and UDP multiple functions are collaborating to create the final communication channel. The local address and port of the server endpoint and the remote address and port of the client endpoint are used to identify the channel. This algorithm first try to retrieve the local address and port of the server endpoint and remote address and port from client endpoint. Then it try to match two endpoints by comparing the local and remote address and port.

**Algorithm 4: Communication Indentification Algorithm for TCP and UDP**

**Input:** two endpoint lists of the dual-trace

**Output:** channel list

```
communications ← Map⟨String, List⟨Communication⟩⟩;
for endpoint1 ∈ endpointList1 do
    bindEvent1 ← get the bind() function call related event from
     endpoint1.openStream;
    connectEvent1 ← get the connect() function call related event from
     endpoint1.openStream;
    for endpoint2 ∈ endpointList2 do
        bindEvent2 ← get the bind() function call related event from
         endpoint2.openStream;
        connectEvent2 ← get the connect() function call related event from
         endpoint2.openStream;
        if socketEvent1! = null AND socketEvent2! = null then
            if bindEvent1! = null AND connectEvent2 == null then
                localServerAddr ← get the serverAddr parameter value from
                 bindEvent1.inputs;
            else if bindEvent2 == null AND connectEvent1! = null then
                remoteServerAddr ← get the serverAddr parameter value from
                 connectEvent1.inputs;
            else
                Break the inner For loop;
            if localServerAddr == remoteServerAddr then
                DataVerified ← Call the corresponding data verification algorithm.
                 communication = New Communication();
                communication.endpoint1 = endpoint1;
                communication.endpoint2 = endpoint2;
                communication.dataMatch ← The output from data verification
                 algorithm;
                communications.add(communication);
return channels;
```

### 2.6.3 Transmitted Data Verification for Named Pipe and TCP by Data Union

As describe in Section1.1.1, the data being received by one endpoint should always equal to or at least is sub string of the data being sent from the other endpoint in a communication for the reliable transmission methods, such as Named Pipe and TCP. So the transmitted data verification algorithm is in data union level. The send data union is retrieved by the conjunction of the input buffer content of the send events in the send stream of an endpoint. The receive data union is retrieved by the conjunction of the output buffer content of the receive events in the receive stream of the other endpoint. The input of this algorithm is the two endpoints from two traces which are being matched in the first level.

**Algorithm 5: Transmitted Verification by Data Union**

**Input:** two endpoints from each trace of the dual-trace

**Output:** send data union and receive data union of two endpoints

**return** *Indicator of if transmitted data union are considered to be identical* $send1 \leftarrow$ empty string;
$send2 \leftarrow$ empty string;
$recv1 \leftarrow$ empty string;
$recv2 \leftarrow$ empty string;
**for** $sendEvent \in endpoint1.sendStream$ **do**
    $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.inputs$;
    $send1.append\,(sendmessage)$;
**for** $sendEvent \in endpoint2.sendStream$ **do**
    $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.inputs$;
    $send2.append\,(sendmessage)$;
**for** $recvEvent \in endpoint1.receiveStream$ **do**
    $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.outputs$;
    $recv1.append\,(sendmessage)$;
**for** $recvEvent \in endpoint2.receiveStream$ **do**
    $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.outputs$;
    $recv2.append\,(sendmessage)$;
**if** $recv1$ *is substring of* $send2$ *AND* $recv2$ *is substring of* $send1$ **then**
    **return** True;
**else**
    **return** False;

### 2.6.4 Transmitted Data Verification for MSMQ and UDP by Data of Events

For the unreliable communication methods, the data packets being transmitted are not delivery and ordering guaranteed. So it is impossible to verify the transmitted data as a whole chunk. Fortunately, the packets arrived to the receivers are always as the original one from the sender. Therefore, we perform the transmitted data verification by single events instead of the whole stream. This algorithm basically goes through the send event list from one endpoint trying to find the matched receive event from the receive event list from the other endpoint. And then calculate the fail packet arrival rate. The fail packet arrival rate should be comparable to the packet lost rate. So we set the packet lost rate as the threshold to determine if the transmitted data can considered to be identical in both directions. The packet lost rate can be various from network to network or even from time to time for the same network. The inputs of this algorithm are the copies of two endpoints from two traces which are being matched and the packet lost rate as the threshold. I use copies instead of original endpoints is that for efficiency I modify the input list directly in the algorithm. The threshold should be an integer. For example if the lost rate is 5%, the threshold should be set as 5.

**Algorithm 6: Transmitted Verification by Data of Events**

**Input:** two endpoint lists of the dual-trace, threshold

**Output:** matched event list of two endpoints

**return** *Indicator of if transmitted data union are considered to be identical*
$sendPktNum1 \leftarrow endpoint1.sendStream.length$;
$sendPktNum2 \leftarrow endpoint2.sendStream.length$;
$recvPktNum1 \leftarrow 0$;
$recvPktNum2 \leftarrow 0$;
$eventMatchs \leftarrow List\langle EventMatch\rangle$;
**for** $sendEvent \in endpoint1.sendStream$ **do**
    $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.inputs$;
    **for** $recvEvent \in endpoint2.receiveStream$ **do**
        $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.outputs$;
        **if** $sendmessage == recvmessage$ **then**
            $recvPktNum1++$;
            $endpoint2.receiveStream.remove\,(recvEvent)$;
            $eventMatch = NeweventMatch\,()$;
            $eventMatchs.add\,(eventMatch)$;
**if** $(sendPktNum1 - recvPktNum1) * 100/sendPktNum1 > threshold$ **then**
    **return** False;
**for** $sendEvent \in endpoint2.sendStream$ **do**
    $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.inputs$;
    **for** $recvEvent \in endpoint1.receiveStream$ **do**
        $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.outputs$;
        **if** $sendmessage == recvmessage$ **then**
            $recvPktNum2++$;
            $endpoint1.receiveStream.remove\,(recvEvent)$;
**if** $(sendPktNum2 - recvPktNum2) * 100/sendPktNum2 > threshold$ **then**
    **return** False;
**return** True;

## 2.7 Data Structures for Identified Communications

In the previous sections, I elaborate all the essential algorithms to identify the communications. The information of identified communications should be organized properly for the further presentation or visualization to the user. In this section, I define the output data structures to fulfil this requirement. There are totally two major data set. The first one is clustered as communications aligning the definition at Section1.2. The second one is clustered by endpoints in the traces. The reason to provide the second data set is due to the false positive errors of the channel identification. The identified endpoint lists of the traces provide more original data information. So with other assistant information and the access of this relatively original information of the dual-trace, the user has more flexibility to analysis the dual-trace. The data structures have been used in the algorithms implicitly.

**Algorithm 7: Data Structure for Identified Communications**

```
1  communications ← Map⟨String, List⟨Communication⟩⟩;        // communication
     clustering
2  traceEndpoints ← Map⟨String, List⟨Endpoint⟩⟩;      // endpoint clustering
3  struct {
4  |   Endpoint endpoint1      // endpoint1 is from trace1 of the dual-trace
5  |   Endpoint endpoint2      // endpoint2 is from trace2 of the dual-trace
6  |   DataMatch dataMatch
7  } Communication
8  struct {
9  |   Int       handle
10 |   Stream    stream
11 } Endpoint
12 union {
13 |   DataUnionMatch    unionMatch        // For data union verification
14 |   List ⟨ EventMatch ⟩  eventMatchs     // For data event verification
15 } DataMatch
16 struct {
17 |   String sData1              // send data union of endpoint1
18 |   String rData1   // receive data union of endpoint1,substring of sData2
19 |   String sData2              // send data union of endpoint2
20 |   String rData2   // receive data union of endpoint2,substring of sData1
21 } DataUnionMatch
22 struct {
23 |   Event      event1              // event1 is from enpoint1
24 |   Event      event2              // event2 is from enpoint2
25 } EventMatch
26 struct {
27 |   List ⟨ Event ⟩  openStream
28 |   List ⟨ Event ⟩  closeStream
29 |   List ⟨ Event ⟩  sendStream
30 |   List ⟨ Event ⟩  receiveStream
31 } Stream
32 struct {
33 |   Int                       lineNum
34 |   Map ⟨ String, String ⟩  inputs
35 |   Map ⟨ String, String ⟩  outputs
36 } Event
```

# Chapter 3

# Additional Information

## 3.1 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.

# 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.