Communication Analysis of Programs by Assembly Level Execution Traces

by

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

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

MASTER OF SCIENCE

in the Department of Computer Science

Communication Analysis of Programs by Assembly Level Execution Traces

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. Daniel German, Supervisor
(Department of Computer Science)

Dr. Margaret-Anne Storey, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Daniel German, Supervisor
(Department of Computer Science)

Dr. Margaret-Anne Storey, Departmental Member
(Department of Computer Science)

ABSTRACT

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

DEDICATION

Just hoping this is useful!

Chapter 1

Introduction

The vulnerability of the software enable the exploitation of the computer or system it is running on. Wherefore, the emphasize placed on computer security particularly in the field of software vulnerabilities increase dramatically along with the grow of the internet. It's important for software developers to build secure applications. Assurance about the integrity and security of the software are expected from the vendors of the software. Unfortunately, building secure software is expensive. The vendors usually comply with their own quality assurance measures which focus on marketable concerns while left security in lower priority or even worse totally ignore it. Therefore, fully relying on the vendor of the software to secure your system and data is unpractical and risky.

Software security review conducted by a third party is usually more convincing and comprehensive. One approach of software security review is software auditing. It is a process of analyzing the code of the software in form of source code or binary. This auditing can uncover some hard to reveal vulnerabilities which might be misused by the hackers. Identification of these security holes can safe the users of the software from putting their sensitive data and business resources at risk.

Most of the software vulnerabilities are caused by malicious data intrusion. So it is valuable to understand how this malicious data trigger the unexpected behaviors of the system. In most of the case this malicious data is injected by an attacker into the system to trigger the exploit. Nonetheless, this malicious data might go through devious path to ultimately triggers an exploitable condition of the system. In some complicated system, several components which are collaborated programs work together to provide service or functionality. In these situation the malicious data might have passed through multiple components of the system and be modified before it reach the vulnerable point of the system. As a consequence, the flow of data throughout the system's different programs is considered to be one of the most important attribute to analysis during the security review.[6]

The data flow among various programs with in the system or across different systems helps to understand how the system work as well as disclose the vulnerabilities of the system as stated before. There are multiple mechanism to grab the data across programs. And the methods for the grasp of this data flow is essential and can affect the analysis result greatly. For instance, packet capture by some sort of sniffers in the network is considered to be insufficient for security problems detection by the experience security engineer from DRDC(research partner of our group). Instead, dynamic analysis of the programs by capturing and analyzing their execution traces with memory accesses is a relatively more accurate method to analyse the data transmitted throughout the programs of the system.

In this research, I developed a method to analysis communications between the programs by the analysis of the execution traces of them. I didn't aim at covering all the communication types but only focus on the data exchanging ones. This method should be able to guide the security engineers to investigate the communications of the programs in the circumstance that they have the captured execution traces and want to understand the interaction behavior of the programs. The research is not specified for vulnerabilities detection but generalized for the comprehension of the behavior of the programs.

1.1 Motivation

This project started with an informal requirement from DRDC for visualizing multiple assembly traces to assist their software and security analysis. The literature review and the conversation with DRDC help to clarify the goal and target of this research. In this section, I discuss the need of performing assembly trace investigation for communication analysis. First I explain why security engineers perform assembly trace analysis. Then I elaborate why they need to perform communication analysis at assembly trace level.

1.1.1 Why Assembly Trace Analysis

Dynamic analysis of program is adopted mainly in software maintenance and security auditing[24], [4], [19]. Sanjay Bhansali et al. claimed that program execution traces with the most intimate detail of a program's dynamic behavior can facilitate the program optimization and failure diagnosis. Jonas Trümper et al. give a example of how tracing can facilitate software-maintenance tasks [21].

The dynamic analysis can be done by using debuggers, however, debuggers would halt the execution of the system and result in a distortion of the timing behavior of the running system

[21]. Instead, tracing a running program with instrumentation would provide more accurate run time behavior information about the system.

The instrumentation of the tracing can be in various level, such as programming language or machine language levels. The choice in some how depends on the accessibility to the application. The application access is divided into five categories with variations: source only, binary only, both source and binary access, checked build, strict black box. Binary only category is common when performing vulnerability research on closed-source commercial software.[6] In this case, assembly level tracing provides the possibility of the security review of the software.

On the other hand, since the binary code is actually what is running on the system, it is more representative of what is actually running than the source code. Some bugs might appear because of a compilation problem or because the compiler optimized away some code that was necessary to make the system secure. The piece of code list below is an example in which the line of code resetting the password before the program end would be optimized away by GNU Compiler Collection(GCC) due to it is not used later. This made the user's password stayed in memory, which is considered as a security flaw. However, by looking at the source code didn't reveal that problem.

Listing 1.1: Password Fetching Example

```
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main(){
  string password ="";
  char ch;
  cout << "Enter password";
  ch = _getch();
  while(ch != 13){//character 13 is enter
    password.push_back(ch);
    cout << '*';
    ch = _getch();
  }
  if(checkPass(password)){
   allowLogin();
  }
  password ="";
}
```

1.1.2 Why Communication Analysis with Assembly Traces

Programs nowadays do not alway work isolated, many software appear as reticula collaborating systems connecting different modules in the network[16] which make the discovery of vulnerabil-

ities even harder. The communication and interaction between modules affect the behavior of the software. Without regarding to the synergy information, analysis of the isolated execution trace on a single computer is usually futile. The tracing data flow process is essential to reviews of both the design and implementation of software.

Many network sniffer such as Wireshark[5] and Tcpdump[20] can help to capture the data flow across the network which make the systematic analysis possible. However, it is claimed as the insufficient method due to the fact that security problems can occur even if the information sent is correct. Therefore, analyzing the communications with transmitted data in instruction and memory access level is a solid way to evaluation the system.

Shameng Wen et al. argued that fuzz testing and symbolic execution are widely applied to detect vulnerabilities in network protocol implementations. They present their model-guided approach to detect vulnerabilities in the network protocol implementation. Their work focus on designing the model which guide the symbolic execution for the fuzz testing but ignoring the analysis of the output, which is the execution traces. [22] Further more, their work focus only on the network protocol implementation but not generalized to all communications, including network protocol and inter process communications.

Besides vulnerabilities detection and security reason, communication analysis with assembly traces can also be a way to learn how the work is performed by the system or sometime validate a specification of it. Our research partner DRDC provided use cases of communication analysis which related to their work with embedded system. These systems often have more than one processor, each specialized for a specific task, that coordinate to complete the overall job of that device. In other cases, the embedded device will work with a normal computer and exchange information with it through some means (USB, wireless, etc.). For instance, the data might be coming in from an external sensor in an analog form, transformed by a Digital Signal Processor (DSP) in a device, sent to a more generic processor inside that device to integrate with other data then send wireless to an external computer. Being able to visualize more than one trace would help them follow the flow of data through the system.

1.2 Approach

The approach I elaborate in this section is not a forthright process. Instead, it is a back and forth one, for example the implementation changed several times with the changes of the model, and the models was modified based on the understanding throughout the implementation. Here I simply my research approach by listing the key factors in each step while ignoring route of it.

This research requires background knowledge in software security, program communication mechanism and implementation, assembly execution traces. I acquire the software security knowledge basically from literature reviews. It helps me to grab the essential concept of software vulnerabilities and their categories, understand some facilities for vulnerabilities detection and software maintenance in the perspective of security. After that, I was convinced that communication analysis in assembly trace level would benefit software security engineers to understand to behavior of the software and detect software vulnerabilities.

In order to analysis the communication of programs, I had to know how the communication works. For this purpose, I started the investigation from a piratical experiment by writing example simple programs with the Windows API and run them locally in my desktop. By understanding their behavior and the reading of the Windows API documentation, I abstracted the communication model which is not operating system specific.

The abstract assembly trace definition was build on the generalization of the trace format provided by our research partner, DRDC. I don't have the access to their home-made assembly tracer which is based on PIN[11]. Fortunately, they provides a comprehensive document about the format of the captured trace and example traces to me. With these, I grasped the constructive view of the assembly execution trace. Further more, in the present of some dynamic software analysis works [9], [14], [18], [1], [2] and [21], it is certain that, some other tools can also capture the required information in assembly level for communication analysis. This supports the generalization of the trace definition and the abstraction of the dual_trace.

The implementation of the prototype and the communication identification algorithms are develop in parallel. The high level identification algorithm and the specific algorithms for named pipe communication methods were abstracted based on the implementation, while the others are developed theoretically.

Among the two experiments, the first simpler one was provided directly by DRDC with their initial requirement, while the second one was designed by me. In both experiments, DRDC conducted the program execution and trace capture on their environment while I performed the analysis locally with Atlantis on my desktop with the captured traces and corresponding .dll.

1.3 Thesis Organization

In Chapter 2, I summarize the related background information and knowledge needed to understand or related to this work including security and vulnerability, program communication mechanisms, program execution trace tools, and Atlantis.

Chapter 3 depicts the model of the communication between two programs. The communication model defines the communication in the context of trace analysis as well as discusses the properties of the communications.

Chapter 4 presents the abstract dual_trace definition. Based on this definition, I developed the communication identification process and the essential algorithms.

To provide more concrete idea, I present, in chapter 5, the implemented communication identification feature prototype. This prototype was built on top of Atlantis[10], an assembly execution trace analysis environment.

In chapter 6, I present two experiments of communication analysis with dual_traces of programs using the implemented prototype. Notably, the result shows the communications are correctly identified.

Finally, In chapter 7, I conclude the result of this research and outline the possible future work.

Chapter 2

Background

In this section, I summarize the background knowledge or information that related to this work. First I generally describe software vulnerability. Second, I discuss the general definition of communication among programs and their categorization. Third, I introduce some tools for assembly level program debugging and analysis. Finally I introduce Atlantis, the existing assembly level execution trace analysis environment, on which the implementation of this work based.

2.1 Software Vulnerability

Software vulnerability detection is one of the use cases of the communication analysis with assembly level execution traces. The understanding of fundamentals of software vulnerability is necessary to comprehend some implicit concepts or design intention through out this thesis.

Vulnerabilities, from the point of view of software security, are specific flaws or oversights in a program that can be exploited by attackers to do something malicious expose such as modify sensitive information, disrupt or destroy a system, or take control of a computer system or program. They are considered to be a subset of bugs. Input and Data Flow, interface and exceptional condition handling where vulnerabilities are most likely to surface in software and memory corruption is one of the most common vulnerabilities. The awareness of these two facts would make the security auditing and vulnerabilities detection have more clear focus. [6]

2.2 Program Communications

Programs can communicate with each other via diverse mechanisms. The communication happens among processes is known as inter-process communication. This refers to the mechanisms an

operating system provides the process to share data with each other. It includes methods such as signal, socket, message queue, shared memory and so on.[8] This communications can happen over network or inside a device. Based on their reliability, the communication methods can be simply divided into two categories: reliable communication and unreliable communication. In this work, communication methods belong to all both categories has been covered. However, I only discuss the message based communication methods while leave the control based communication, like remote function call for the future.

2.3 Program Execution Tracing in Assembly Level

The communication analysis discuss throughout this thesis is based on the assembly traces. Thurs capturing of the execution traces became a prerequisite of this work. DRDC has its own home-made tracer, the traces from which are used in the experiments of this research. However, the model and algorithms developed in this research is not limited with this specific home-made tracer. Any tracer that can capture sufficient information according to the model can serve this purpose.

There are many tools that can trace a running program in assembly instruction level. IDA pro [7] 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 [12] to provide more sufficient features for ”run-trace” visualization. PIN[11] 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 [3] and OllyDbg[23] also provide the debugging and tracing functionality in assembly level.

2.4 Atlantis

Atlantis is a trace analysis environment developed in Chisel. It can support analysis for multi-gigabyte assembly traces. There are several features distinct it from all other existing tools and make it particularly successful in large scale trace analysis. These features are 1) reconstruction and navigation the memory state of a program at any point in a trace; b) reconstruction and navigation of system functions and processes; and c) a powerful search facility to query and navigate traces. The work of this thesis is not a extension of Atlantis. But it take advantages of Atlantis by reusing it existing functionality to assist the dual_trace analysis.

Chapter 3

Communication Modeling

In this chapter, I depict the model of the communication of two running programs from the trace analysis point of view. The modeling is based on the investigation of some common used communication methods. But the detail of the communication methods will be discussed later in the algorithm and implementation chapters. This chapter only present the abstract communication model regarding to the two communication categories: reliable and unreliable communications.

3.1 Communication Methods Categorization

In general, there are two types of communication: reliable and unreliable in the terms of their reliability of data transmission. A reliable communication guarantees the data being sent by one endpoint through the channel is always received losslessly and in the same order in the other endpoint. On contrast, an unreliable communication does not guarantee the data being sent 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. An endpoint is an instance in a program at which a stream of data is sent or a stream of data is received or both (e.g. socket handle of TCP or a file handle of the named pipe). A channel is a conduit connected two endpoints through which data can be sent and received. Table3.1 gives examples of communication methods fall in these two categories.

Table 3.1: Communication Method Examples in Two Categories

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

3.2 Communication Model

The communication of two programs is defined in this section. The communication in this work is data transfer activities between two running programs through a specific channel. Some collaborative activities between the programs such as remote procedure call is out of the scope of this research. Communication among multiple programs (more than two) is not discussed in this work. The channel can be reopened again to start new communications after being closed. However, the reopened channel is considered 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 in the real world and they are compatible to this communication definition.

3.2.1 Communication Definition

In the context of a dual_trace, a communication is a sequence of data transmitted of two endpoints through a communication channel. The endpoints connect to each other using the identifier of this channel. We therefore defined a communication c as a triplet:

$c =< ch, e_0, e_1 >$

where e_0 and e_1 are endpoints while ch is the communication channel (e.g. a named piped located at /tmp/pipe).

From the point of view of traces, the endpoints e_0 and e_1 are defined in terms of three properties: the handle created within a process for the endpoint for subsequent operations(e.g. data send and receive), the data stream received and the data stream sent. Therefore, I define an endpoint e as a triplet:

$e =< handle, d_r, d_s >$

where $handle$ is the handle identifier of the endpoint, d_r and d_s are data streams. A data stream is a sequence of sent packages or a sequence of received packages. Each package pk contains data that is being sent or received (its payload). Hence, we can define a data stream d as a sequence of n packages:

$d = (pk_1, pk_2, ..., pk_n)$

Note: This is the sequence of packages as seen from the endpoint and might be different than the sequence of packages seen in the other endpoint, specially where there is package reordering, loss or duplication.

Each package pk has two attributes:

- *Relative time(it was sent or received):* In a trace, we do not have absolute time for an event. However, we know when an event (i.e. open, close, sending or receiving a package) has happened with respect to another event. I will use the notation

  $time(pk)$

  to denote this relative time. Hence,

  if $i < j$, then $time(pk_i) < time(pk_j)$

- *Payload:* Each package has a payload (the data being sent or received). I use the notation

  $pl(pk)$

  to denote this payload.

3.2.2 Communication Properties

The properties of the communications can be described based on the definition of the communication.

Properties of reliable communication:

A reliable communication guarantees that the data sent and received between a package happens without loss and in the same order.

For a given data stream, I define the data in this stream as the concatenation of all the payloads of all the packages in this stream, in the same order, and denote it as $data(d)$.

Given $d =< pk_1, pk_2, ..., pk_n >$, $data(d) = pl(pk_1) \cdot pl(pk_2) \cdot \ldots \cdot pl(pk_n)$

- *Content Preservation:* for a communication:

  $c =< ch, < h_0, dr_0, ds_0 >, < h_1, dr_1, ds_1 >>$

  the received data of an endpoint should always be a prefix of (potentially equal to) the sent data of the other:

  $data(dr_0)$ is a prefix of $data(ds_1)$ and

$data(dr_1)$ is a prefix of $data(ds_0)$

- *Timing Preservation:* at any given point in time, the data received by an endpoint should be a prefix of the data that has been sent from the other:

  for a sent data stream of size m, $ds =< pks_1, pks_2, ...pks_m >$ that is received in data stream of size n, $dr =< pkr_1, pkr_2, ...pkr_n >$

  for any $k \in 1..n$, there must exist $j \in 1..m$ such that:

  pks_j was sent before pkr_k was received: $time(pks_j) < time(pkr_k)$

  and

  $data(< pkr_1, pkr_2, ..., pkr_k >)$ is a prefix of $data(< pks_1, pks_2, ..., pks_j >)$

  In other words, at any given time, the recipient can only receive at most the data that has been sent.

Properties of unreliable communication:

In unreliable communication the properties are not concerned in the concatenation of packages. Instead, each package is treated as independent of each other.

- *Content Preservation:* a package that is received should have been sent:

  for a sent data stream of size m, $ds =< pks_1, pks_2, ...pks_m >$ that is received in data stream of size n, $dr =< pkr_1, pkr_2, ...pkr_n >$

  for any $pkr_j \in dr$ there must exist $pks_i \in ds$

  We will say that the pkr_j is the matched package of pks_i, and vice-versa, pks_i is the matched package of pkr_j, hence

  $match(pkr_j) = pks_i$ and

  $match(pks_i) = pkr_j$

- *Timing Preservation:* at any given point in time, packages can only be received if they have been sent

  for a sent data stream of size m, $ds =< pks_1, pks_2, ...pks_m >$ that is received in data stream of size n, $dr =< pkr_1, pkr_2, ...pkr_n >$

  for any $k \in 1..n$, $time(match(pkr_j)) < time(pkr_j)$

  In other words, the match of the received package must has been sent before it is received.

In the following two examples, h_0 and h_1 are the handles of the two endpoints e_0 and e_1 of the communications. ds_0, dr_0 and ds_1, dr_1 are the data streams of the endpoints e_0 and e_1. The payloads are the strings represented in blue and red in the figures.

Figure3.1 is an example of the reliable communication.

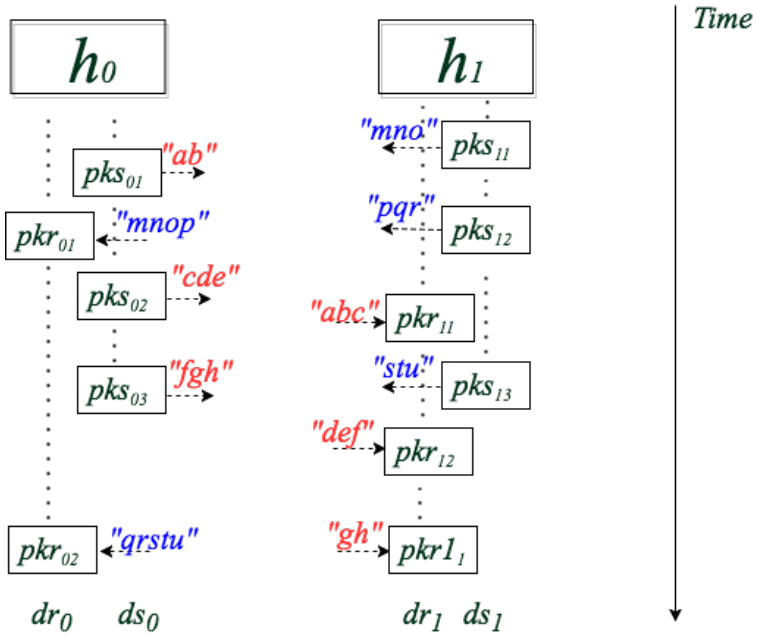

Figure 3.1: Example of Reliable Communication

In this example, the payloads of the packages are:
$pl(pks_{01}) =$ "ab", $pl(pks_{02}) =$ "cde", $pl(pks_{03}) =$ "fgh";
$pl(pkr_{11}) =$ "abc", $pl(pkr_{12}) =$ "def", $pl(pkr_{13}) =$ "gh" .
and
$pl(pks_{11}) =$ "mno", $pl(pks_{12}) =$ "pqr", $pl(pks_{13}) =$ "stu";
$pl(pkr_{01}) =$ "$mnop$", $pl(pkr_{02}) =$ "$qrstu$".
on the other direction.
Their properties:
$pl(pks_{01}) \cdot pl(pks_{02}) \cdot pl(pks_{03}) = pl(pkr_{11}) \cdot pl(pkr_{12}) \cdot pl(pkr_{13}) =$ "$abcdefgh$"
and
$pl(pks_{11}) \cdot pl(pks_{12}) \cdot pl(pks_{13}) = pl(pkr_{01}) \cdot pl(pkr_{02}) =$ "$mnopqrstu$".
satisfy the content preservation.
The relative time relationship of the packages are:
$time(pks_{01}) < time(pks_{02}) < time(pkr_{11}) < time(pks_{03}) < time(pkr_{12}) < time(pkr_{13})$;

$time(pks_{11}) < time(pks_{12}) < time(pkr_{01}) < time(pks_{13}) < time(pkr_{02})$.

The fact that

$pl(pkr_{01}) =$ "$mnop$" is the prefix of $pl(pks_{11}) \cdot pl(pks_{12}) =$ "$mnopqr$",

$pl(pkr_{01}) \cdot pl(pkr_{02}) =$ "$mnopqrstu$" is the prefix of (in this case is identical to) $pl(pks_{11}) \cdot pl(pks_{12}) \cdot pl(pks_{13}) =$ "$mnopqrstu$",

$pl(pkr_{11}) =$ "abc" is the prefix of $pl(pks_{01} \cdot pl(pks_{02}) =$ "$abcde$",

$pl(pkr_{11}) \cdot pl(pkr_{12}) =$ "$abcdef$" and $pl(pkr_{11}) \cdot pl(pkr_{12}) \cdot pl(pkr_{13}) =$ "$abcdefgh$" are the prefix of $pl(pks_{01}) \cdot pl(pks_{02}) \cdot pl(pks_{03}) =$ "$abcdefgh$"

satisfy the timing preservation.

Figure3.2 is an example of the unreliable communication.

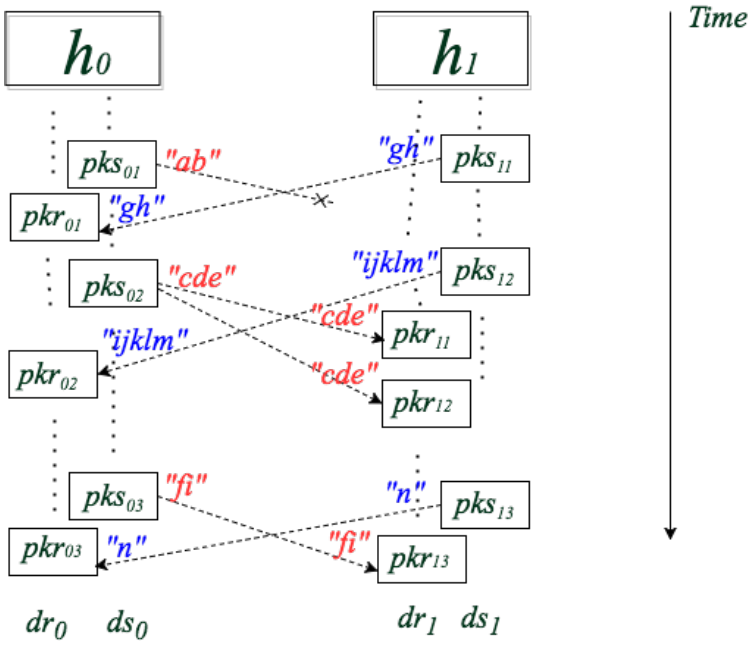

Figure 3.2: Example of Unreliable Communication

In this example, the content preservation of the unreliable communication are satisfied since:

$pkr_{11} = pks_{02} =$ "cde";

$pkr_{12} = pks_{02} =$ "cde";

$pkr_{13} = pks_{03} =$ "fi";

$pkr_{01} = pks_{11} =$ "gh";

$pkr_{02} = pks_{12} =$ "$ijklm$";

$pkr_{03} = pks_{13} =$ "n".

while the timing preservation of the unreliable communication are satisfied since:

$time(pkr_{11}) > time(pks_{02})$;
$time(pkr_{12}) > time(pks_{02})$;
$time(pkr_{13}) > time(pks_{03})$;
$time(pkr_{01}) > time(pks_{11})$;
$time(pkr_{02}) > time(pks_{12})$;
$time(pkr_{03}) > time(pks_{13})$;

Chapter 4

Communication Identification

The goal of this work is to identify the communications from the dual_trace. A dual_trace is a pair of assembly level execution traces of two interacting programs. In this chapter, I discuss the characteristics of the execution trace and give the abstract definition of the dual_trace and the execution trace. Based on the communication model in Chapter3, I designed the analysis approach for communication identification from the dual_trace. This approach includes the algorithms and identification process. For all traces comply with this abstract trace definition, this analysis approach can be applied for the communication identification.

4.1 Dual_Trace

A dual_trace consists of two assembly level execution traces of two interacting programs. There is no timing information of these two traces which means we don't know the timing relationship of the events of one trace with respect to the other. However the captured instructions in the trace are ordered in execution sequence. An execution trace consist of a sequence of instruction lines. Each instruction line contains the executed instruction, the changed memories, the changed registers, execution information. The execution information indicates the execution type which can be: Instruction, System call entry, System call exit, etc. For the execution type of system call entry and system call exit, system call Id is given in this information. With the system call id and the provided .dll files, the called system function name can be obtained.

A dual_trace is formally defined as :

$dual\_trace = \{trace_0, trace_1\}$

where $trace_0$ and $trace_1$ are two assembly execution traces.

A trace is a sequence of executed instruction line. Hence, I define a $trace$ as a sequence of n

instruction lines:

$trace = (l_1, l_2, ..., l_n)$

Each instruction line, l is a tuple:

$l =< ins, mch, rch, exetype, syscallInfo >$

where ins is the instruction, mch is the memory changes, rch is the register changes, $exetype$ is the execution type which can be instruction, system call entry, system call exit, and other types which are not concerned in this work, $syscallInfo =< exeName, offset >$ only appear when $exetype$ is system call entry or system call exit. $exeName$ is the executable file name(e.g. .dll and .exe), while $offset$ is the offset of the system function in this executable file.

Figure4.1 is an example of a piece of execution trace complying to this definition.

Line	Instruction	Memory Changes	Registster Changes	Execution Type	System Call Info
l_{1499}	lea rcx, ptr [rip+0x3cfe0]		ECX F8A0AAA8	Instruction	
l_{1500}	xor r9d, r9d			Instruction	
l_{1501}	mov edx, 0xc0000000		EDX C0000000	Instruction	
l_{1502}	mov dword ptr [rsp+0x20], r8d	00000000001DF490 00000003 00000000	RSP 00000000 001DF470	Instruction	
l_{1503}	call qword ptr [rip+0x3a97d]	00000000001DF470 000007FE F89CDADB	RSP 00000000 001DF468	System Call	kernel32.dll+77270D10
l_{1504}	mov qword ptr [rsp+0x8], rbx	00000000001DF470 00000000 00000001	EBX 00000001	Instruction	
l_{1505}	mov qword ptr [rsp+0x10], rbp	00000000001DF478 00000000 00000000	EBP 00000000 ESP 001DF468	Instruction	
l_{2057}	add rsp, 0x20		ESP 001DF2A0	Instruction	
l_{2058}	pop rbx		RBX 00000000 001DF3E8 RSP 00000000 001DF2A8	Instruction	
l_{2059}	ret		RSP 00000000 001DF2B0	System Call Exit	kernel32.dll+77281903
l_{2060}	mov eax, dword ptr [rsp+0x54]		EAX 00000000	Instruction	

Figure 4.1: An example trace

Figure4.2 is an example of the information decoded from a executable file kernal32.dll. From this example we can see if an instruction line is a system call entry and its $syscallInfo =< kernal32.dll, 0x11870 >$, this line is a function call to $copyFileExW$.

Offset	Name
0x47b90	CopyContext
0x95690	CopyFileA
0x95440	CopyFileExA
0x11870	CopyFileExW
0x955c0	CopyFileTransactedA
0x954f0	CopyFileTransactedW
0x8950	CopyFileW
0x8fa60	CopyLZFile
0x6fcd0	CreateActCtxA
0x1a170	CreateActCtxW
0x610c0	CreateBoundaryDescriptorA
0x4b7b0	CreateBoundaryDescriptorW
0x413f0	CreateConsoleScreenBuffer
0x4ca50	CreateDirectoryA
0x8a220	CreateDirectoryExA
0x892e0	CreateDirectoryExW
0x8a370	CreateDirectoryTransactedA
0x8a2a0	CreateDirectoryTransactedW
0xa220	CreateDirectoryW
0x105c0	CreateEventA

Figure 4.2: Information of kernal32.dll

4.2 Function Call Event Reconstruction Algorithm

In last section, I defined the assembly execution trace. As to this definition, it is possible to recognize the function call entry and function call exit from the trace. In this section, I define the function call event and present the algorithm to reconstruct the function call events from the assembly execution trace.

There would be lots of function calls in an execution trace. However, most of them are not of interest. This algorithm will only reconstruct the function call events of a specific communication method. To be able to identify and reconstruct the functions of interest, the communication description is required. The communication description is a set of function descriptions of a communication method and defined as:

$cdes = \{fdes_1, fdes_2, ..., fdes_p\}$

Each element $fdes$ can be defined as:

$fdes = \{name, type, pdes\}$

where, $name$ is the function name, $type$ is the function type which can be one of the four types: $open$, $close$, $send$ and $receive$. The function type is not used in this algorithm but will be inherited by the reconstructed function call events and used in the event stream extraction algorithm. $pdes$ is the parameter description illustrates how the registers and memory contents map to a given function call and the list of its parameter of interest(you might not care for all parameters).

Table4.1 is an example of a function description. In this example, the function name is ReadFile, it is a function for data receiving, so that function type is $receive$. The parameter description includes the concerned parameters, which are Handle, RecvBuffer and MessageLength. The Handle is an input parameter which is a value stored in the register RCX. The RecvBuffer is an address for the input message stored in the register RAX. The MessageLength is a output value stored in register R9. The value of the input parameters can be retrieved from the memory state on the function call instruction line while the value of the output parameters can be retrieved from the memory state on the function return instruction line. If a parameter is an address instead of value, the address should be retrieved first, then the retrieved address should be used to find the buffer content in the memory state. The function description requires the understanding of the calling convention of the operating system. The Microsoft x64 calling convention can be found in Appendix A. More examples of communication method descriptions will be given in Chapter5.

Table 4.1: An example of a function description

Name	**Type**	**Parameter Description**			
		Parameter	**Register/Stack Position**	**In or Out**	**Buffer Or Value**
ReadFile	receive	Handle	RCX	In	Value
		RecvBuffer	RDX	In	Address
		MessageLength	R9	Out	Value

With the communication method description and the execution trace as input, the function call event reconstruction algorithm outputs a size m sequence of function call events which can be defined as:

$etr = (ev_1, ev_2, ..., ev_m)$

A function call event ev in etr is defined as a triplet:

$ev =< funN, paras, type >$

where $funN$ is the function name, $paras$ includes all the input/output parameters with the parameter name and value, and $type$ is the event type which inherit from the function description and can be one of the four types: $open$, $send$, $receive$ and $close$.

Note: If the parameter is an address, the value is the string from the buffer pointed by this address instead of the address value.

An example of a sequence of function call events as the output of this algorithm is shown in Listing4.1.

Listing 4.1: Example of etr

```
{funN:CreateNamedPipe, paras:{Handler:18, FileName:mypipe}, type:open},
{funN:CreateNamedPipe, paras:{Handler:27, FileName:Apipe}, type:open},
{funN:WriteFile, paras:{Handler:27, RecvBuffer:Message1, MessageLength:9}, type:send},
{funN:WriteFile, paras:{Handler:27, RecvBuffer:Message2, MessageLength:9}, type:send},
{funN:ReadFile, paras:{Handler:27, SendBuffer:Message3, MessageLength:9}, type:receive},
{funN:CloseHandle, paras:{Handler:27}, type:close},
{funN:CloseHandle, paras:{Handler:18}, type:close}
```

Algorithm 1 shows the detail of the function call event reconstruction algorithm. This algorithm is designed to reconstruct the function call events of a communication method. If multiple communication methods are under investigated, this algorithm can be run multiple times to achieve the goal. Since the function descriptions usually contain a small number of concerned functions compared to the instruction line number in the execution trace, the time complexity of this algorithm is $O(N)$, N is the instruction line number of the trace.

Algorithm 1: Function Event Reconstruction Algorithm

Input: $trace, cdes$

Output: etr

```
1  etr ← ∅
2  Emulate the Execute of each instruction line of the trace;
3  if The instruction is a call to a function in the cdes then
4      Create an new function call event ev;
5      ev.funN ← fdes.name;
6      ev.type ← fdes.type;
7      Append all the input parameters of interest to ev.paras;
8      Continue the emulation until the function call return line;
9      Append all the output parameters of interest to ev.paras;
10     etr.append(ev);
11 return etr;
```

4.3 Channel Open Mechanisms

The channel open mechanism affect the event stream identification and event stream matching strategy. So I discuss them before presenting those algorithms. The channel open mechanism of named pipe and message queue is relatively simple. In windows implementation, only one function call is related to the handle identification of the stream. However, for TCP and UDP the mechanism is complicated.

4.3.1 Named Pipe Channel Open Mechanisms

For Named Pipe communication method, 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 of a named pipe returns the handle of that pipe. So the identification of the stream only need to identify the pipe creation function call on the server side and the pipe connection function call on the client side. The handle returned by these function calls will be used later when data is being sent or received to a specified pipe. Figure4.3 exemplify the channel set up process for a Named Pipe communication in Windows.

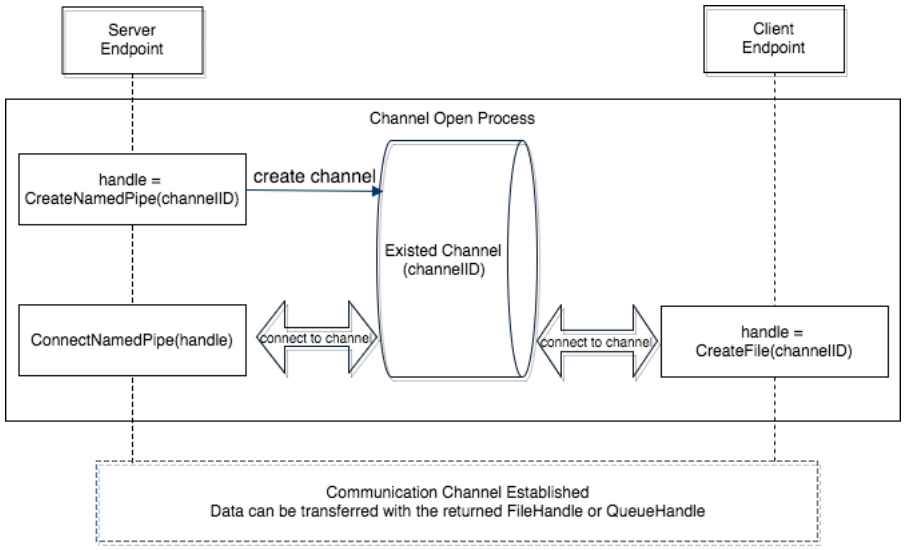

Figure 4.3: Channel Open Process for a Named Pipe in Windows

4.3.2 Message Queue Channel Open Mechanisms

For the Message Queue communication method, the endpoints of the communication can create the queue or use the existing one. However, both endpoints have to open the queue before they access it. The handle returned by the open queue function will be used later when messages are being sent or received to identify the queue. Figure4.4 exemplify the channel set up process for a Message Queue communication in Windows.

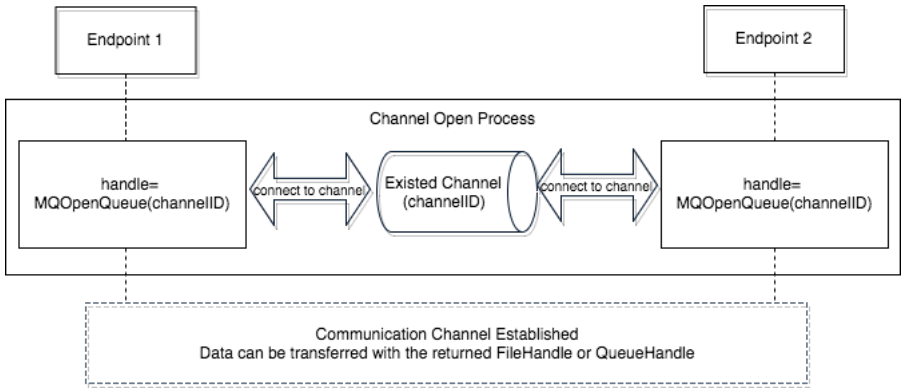

Figure 4.4: Channel Open Process for a Message Queue in Windows

4.3.3 UDP and TCP Channel Open Mechanisms

For UDP and TCP communication methods, the communication channel is set up by both endpoints of UDP and TCP channels. The function *socket* should be called to create their own socket on both endpoints. After the socket handles 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. Figure4.5 exemplify the channel open process for TCP and UDP in Windows.

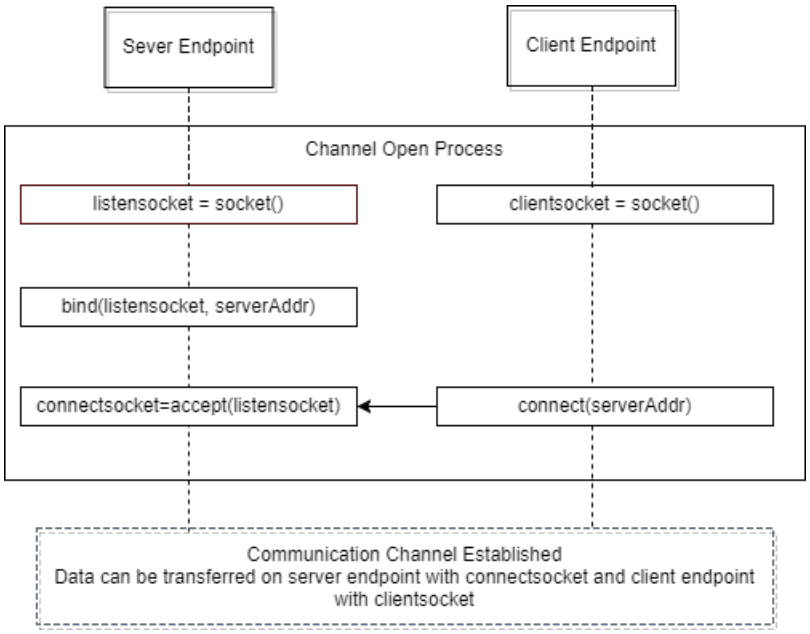

Figure 4.5: Channel Open Model for TCP and UDP in Windows

4.4 Event Stream Extraction Algorithm

The function call events in the etr may belong to different endpoint. An event stream is a size k sequence of function call events corresponding to an endpoint of a communication which can be defined as:

$s = (ev_1, ev_2, ..., ev_k)$

s contains the channel open function call events, data send function call events, data receive function call events and channel close function call events. According to the channel open mechanisms discussed in Section4.3, the identifier of the channel and the handle of the endpoint can be retrieved from the channel open function call events, while the data streams of the endpoint can be received from send function call events and data receive function call events. So that, given the stream s, the endpoint $e =< handle, d_r, d_s >$ and the ch of a communication c defined in a communication in Chapter3 can be retrieved.

Note: the definition of ev in s is identical to that in etr. However, the event numbering of s is different from etr. For example, ev_1 in s and ev_1 in etr might be different events.

The event stream extraction algorithm is designed to extract the streams from the sequence of function call events. In this algorithm, the stream is need to be identified by the channel open function calls first. Then all other events will be added to the stream.

The input of this algorithm is etr from the function event reconstruction algorithm in Algorithm 1. Since the events in etr are reconstructed in sequence of the instructions which are order by the time of occurrence, the events are implicitly sorted by time of occurrence.

The output of this algorithm is a set of stream of size p, which can be defined as:

$str = (s_1, s_2, ..., s_p)$

As to the channel open mechanisms, two different algorithms are designed, one is for Name pipe and Message Queue, while the other is for TCP and UDP.

Event Stream Extraction Algorithm for Named Pipe and Message Queue

This algorithm is designed for extraction of the streams for Named Pipe and Message Queue. Since for each endpoint of the communication, only one channel open function call is needed to identify the endpoint, it is simple to identify the stream once channel open function call event that create the endpoint handle is found.

The same handle may be reused by other endpoint. However, reuse of the handle would not happen before this stream was closed by the channel close function call event. Therefore, before the detection of the channel close function call, if a new channel open function call with the same returned handle is detected, the second channel open will be ignored. This algorithm handle this by having the $tempstream$ set to keep track of the streams that are still open. Once the stream was closed, the handle can be used by another stream. The time complexity of this algorithm is $O(N)$, N is the number of events in the trace.

Algorithm 2: Event Stream Exatraction Algorithm for Named Pipe and Message Queue

Input: etr

Output: str

```
1  str ← ∅
2  tempstreams ← ∅ for ev ∈ etr do
3      h ←the handle identifier from ev.paras;
4      if ev.type = open then
5          if tempstreams[h] not exist then
6              tempstreams[h] ← a new s;
7              tempstreams[h].append(ev);
8      if ev.type = send Or ev.type = receive then
9          if tempstreams[h] exist then
10             tempstreams[h].append(ev);
11     if ev.type = close then
12         if tempstreams[h] exist then
13             tempstreams[h].append(ev);
14             str.append(tempstreams[h]);
15             remove tempstreams[h] from tempstreams;
16 return str;
```

Event Stream Extraction Algorithm for TCP and UDP

This algorithm is designed for extraction of the event streams for TCP and UDP. In the channel open stage, a socket handle created by function call of $socket$ in both client and server. In the server side, this created socket is only used for listening to the client's connection. The listening is accomplished by calling the function $accept$. The input of the $accept$ function call is the listening socket handle, and the output of it is a new data transmission socket handle. This two socket handles are considered to be two handles for two streams, the stream identified by the listening handle is called the parent stream and the one identified by the data transmission handle is called the child stream. However the events in the parents stream contain the information needed for event stream matching algorithm later, so the children stream will inherit all the events from their parents. The time complexity of this algorithm is also $O(N)$, N is the number of events in the

trace.

Algorithm 3: Event Stream Exatraction Algorithm for TCP and UDP

Input: etr

Output: str

```
1  str ← ∅
2  tempstreams ← ∅
3  for ev ∈ etr do
4      if ev.funN = socket then
5          h ← the handle identifier from ev.paras;
6          if tempstreams[h] not exist then
7              tempstreams[h] ← a new s;                    // new a stream
8              tempstreams[h].append(ev);
9      if ev.funN = bind or ev.funN = connect then
10         h ← the handle identifier from ev.paras;
11         if tempstreams[h] exist then
12             tempstreams[h].append(ev);
13     if ev.funN = accept then
14         h ← the input listening handle in ev.paras;    // handle of parent stream
15         hc ← the output data transfer handle in ev.paras; // handle of child stream
16         if tempstreams[h] exist then
17             if tempstreams[hc] not exist then
18                 tempstreams[hc] ← a new s;     // a new stream for the child
19             tempstreams[hc].append(tempstreams[h]); // append parent's events
20             tempstreams[hc].append(ev);        // append the current event
21     if ev.type = send Or ev.type = receive then
22         h ← the handle identifier from ev.paras;
23         if tempstreams[h] exist then
24             tempstreams[h].append(ev);
25     if ev.type = close then
26         h ← the handle identifier from ev.paras;
27         if tempstreams[h] exist then
28             tempstreams[h].append(ev);
29             str.append(tempstreams[h]);
30             remove tempstreams[h] from tempstreams;
31 return str;
```

4.5 Event Stream Matching Algorithm

The function event extraction algorithm and the event stream extraction algorithms all work on a single execution trace. To identify the communications from the dual_trace, I need to match two event streams out of those extracted streams from each trace of the dual_trace. The stream matching algorithm in this section is designed for this purpose. The inputs of the stream matching algorithm are two set of streams str_0 and str_1 which are output by the event stream extraction algorithm. The output of this algorithm is the matching set. Each matched item in this set contains two streams.

This matching algorithm is not fully reliable. There are two situations which false negative error might emerge. Take Named Pipe for example, the first situation is multiple(more than two) interacting programs shared the same file as their own channel. Even though the channels are distinct for each communication, but the file 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 streams found. In each client trace, there is one stream found. For the dual_trace of server and client1, there will be two possible identified communications, one is the real communication for server and client1 while the other is the false negative error actually is for server and client2. The stream in client1's trace will be matched by two streams 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. The matching is based on the identifiers. So that in this case, there will be two matching. Similar situations can also happen in Message Queue, TCP and UDP communication methods. The data stream verification algorithm discussed in Section4.6 can reduce the false negative errors.

The stream 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 streams.

The following two subsections discuss the two matching algorithms, one is for Named Pipe and Message Queue, while the other is for TCP and UDP. For both of the algorithms, the time complexity of this algorithm is $O(N * M)$, N and M are the number of streams in both traces.

4.5.1 Event Stream Matching Algorithm for Named Pipe and Message Queue

For Named Pipe and Message Queue, the first function call event ev_1 in s is the channel open function which is meaningful for endpoint handle identification. The channel identifier parameter can be found in the $ev_1.paras$. The identifier for Named Pipe is the file name of the pipe while for Message Queue is the queue name. This algorithm finds out all the possible communications regardless some of them might be false negative errors. The input str_0 and str_1 are two set of streams from the two traces of the dual_trace. The output ms is a set of two matching streams.

Algorithm 4: Event Stream Matching Algorithm for Named Pipe and Message Queue

Input: str_0, str_1

Output: ms

$ms \leftarrow \emptyset$

for $s_0 \in str_0$ **do**

    /* The first event is the open event */

    $id_0 \leftarrow$ get the channel identifier from $str_0[0].paras$;

    for $s_1 \in str_1$ **do**

        $id_1 \leftarrow$ get the channel identifier from $str_1[0].paras$;

        if $id_0 = id_1$ **then**

            $c.s_0 = s_0$;

            $c.s_1 = s_1$;

            $ms.append(c)$;

return ms;

4.5.2 Event Stream Matching Algorithm for TCP and UDP

For TCP and UDP, multiple function calls collaborate 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 gets the local address and port of from the events in the stream of the server and remote address and port from the events in the stream of the client. Then it tries to match two streams by comparing the local and remote address and port.

Algorithm 5: Event Stream Matching Algorithm for TCP and UDP

Input: str_0, str_1

Output: cs

$cs \leftarrow \emptyset$

for $s_0 \in str_0$ **do**

  $socketev_0 \leftarrow$ the *socket* function call event from str_0;

  $bindev_0 \leftarrow$ the *bind* function call event from str_0;

  $connectev_0 \leftarrow$ the *connect* function call event from str_0;

  for $s_1 \in str_1$ **do**

    $socketev_1 \leftarrow$ the *socket* function call event from str_1;

    $bindev_1 \leftarrow$ the *bind* function call event from str_1;

    $connectev_1 \leftarrow$ the *connect* function call event from str_1;

    // *socket* function call event exists for both server and client streams, *bind* function call event only exist for server streams while *connect* function call event only exist for client streams

    if $socketev_0 != null$ *AND* $socketev_1 != null$ **then**

      // s_0 is a sever stream, s_1 is a client stream

      if $bindev_0 != null$ *AND* $connectev_1 != null$ **then**

        $localServerAddr \leftarrow$ get serverAddr from $bindev_0.paras$;

        $remoteServerAddr \leftarrow$ get serverAddr from $connectev_1.paras$;

      // s_1 is a sever stream, s_0 is a client stream

      else if $bindev_1 != null$ *AND* $connectev_0 != null$ **then**

        $localServerAddr \leftarrow$ get serverAddr from $bindev_1.paras$;

        $remoteServerAddr \leftarrow$ get serverAddr from $connectev_0.paras$;

      if $localServerAddr = remoteServerAddr$ **then**

        $c.s_0 = s_0$;

        $c.s_1 = s_1$;

        $cs.append(c)$;

return cs;

4.6 Data Stream Verification Algorithm

The data stream verification aims to verify if the data streams of the matched event streams align with the communication properties of the communication model in Chapter3. The data transfer characteristics divided the communications into reliable and unreliable categories. Named Pipe and TCP fall in the reliable category while Message Queue and UDP fall in the unreliable one. The properties of the models consists of content preservation and timing preservation. So that the verification should cover both preservations:

- verify the content preservation of the data in the matched streams.
- verify the timing preservation of the data in the matched streams.

To verify the timing preservation, the relative time of the events in both streams is needed. Unfortunately, we can only determine the relative time with in a stream but not crossing two streams. So that it's impossible to verify the timing preservations no matter for reliable or unreliable communications. The verification algorithms discussed in this section will only cover the content preservation.

The inputs of the data stream verification algorithms are two preliminary matched streams s_0 and s_1. The output is the a boolean indicating if the streams satisfy the content preservation. All communications don't satisfy the content preservation should be excluded.

For each communication method the verification of the corresponding preservation is applied, That is, for Named Pipe and TCP, the reliable communication preservation need to be verified and for Message Queue and UDP, the unreliable communication preservation need to be verified. The following sub sections present the versification algorithms for these four communication methods. In each sub section, I discuss the data transfer properties and scenarios of the communication method and then present the developed verification algorithm.

4.6.1 Data Stream Verification Algorithm for Named Pipe

A named pipe provides FIFO communication mechanism for inter-process communication. It can be a one-way or a duplex pipe. [13]

The basic data transfer characteristics of Named Pipe are:

- Bytes are received in order
- Bytes sent as a segment can be received in multiple segments(the opposite is not true)

- No data duplication
- If a sent segment is loss, all the following segments will lost(this happen when the receiver disconnect from the channel)

Based on these characteristics, the data transfer scenarios of Named pipe can be exemplified in Figure4.6.

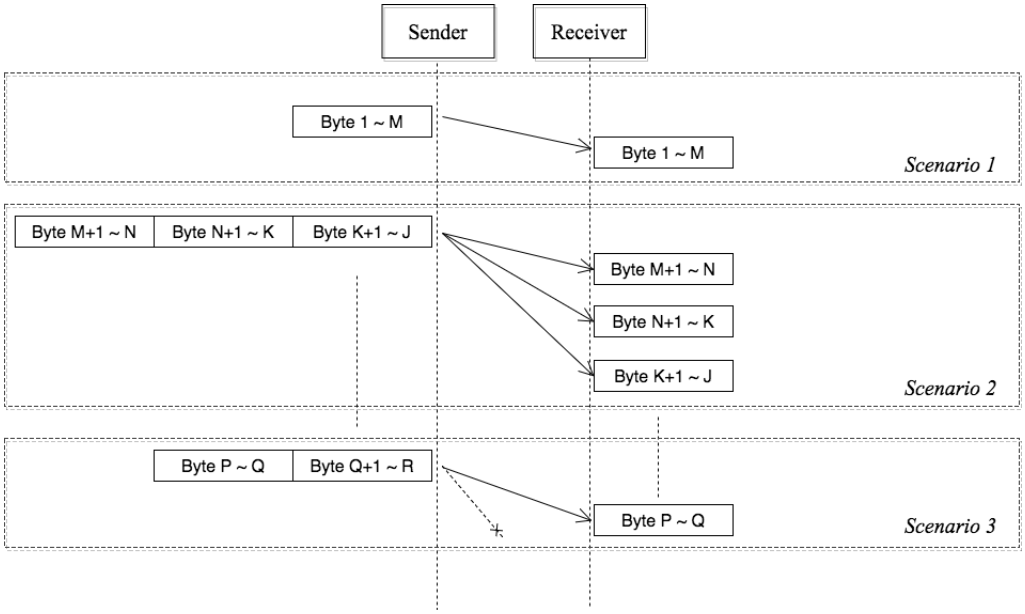

Figure 4.6: Data Transfer Scenarios for Named Pipe

The content preservation verification is trivial as comparing the concatenation of the packet content of the sent events in a stream to the the concatenation of the packet content of the receive events in the other stream, which is presented in Algorithm 4.5. Since the concatenation need to go through the events in the streams, the time complexity of this algorithm is also $O(N)$, N is the total number of data transfer events in the two streams.

Algorithm 6: Data Stream Verification of Named Pipe

Input: s_0, s_1

return $satisfied$

$send_0 \leftarrow$ concatenation of the payload of send function call events in s_0;

$send_1 \leftarrow$ concatenation of the payload of send function call events in s_1;

$receive_0 \leftarrow$ concatenation of the payload of receive function call events in s_0;

$receive_1 \leftarrow$ concatenation of the payload of receive function call events in s_1;

if $receive_1$ *is prefix of* $send_0$ *AND* $receive_0$ *is prefix of* $send_1$ **then**

    return True;

else

    return False;

4.6.2 Data Stream Verification Algorithm for 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 re-transmission 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 re-transmitted)
- No data duplication
- Bytes sent in packet and received in packet, no re-segmentation

Based on these characteristics, the data transfer scenarios of TCP can be exemplified in Figure4.7.

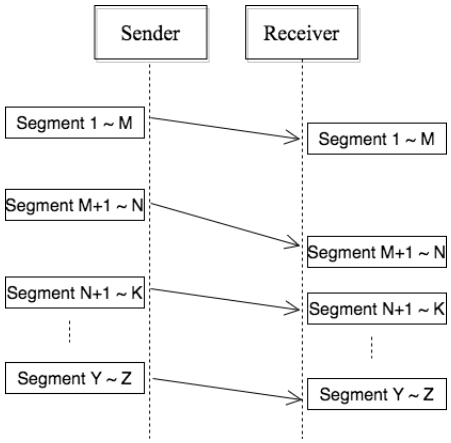

Figure 4.7: Data Transfer Scenarios for TCP

Regrading to the data transfer properties of TCP, the verification can be restricted to packet to packet. If all the send and receive events of the two streams can be matched, we can assert that the content preservation are satisfied. The verification algorithm of TCP is presented in Algorithm 7. The time complexity of this algorithm is also $O(N)$, N is the number of data transfer events in a stream.

Algorithm 7: Data Stream Verification of TCP

Input: s_0, s_1
return $satisfied$
$sends_0 \leftarrow$ all sent events of s_0 in sequence;
$sends_1 \leftarrow$ all sent events of s_1 in sequence;
$receives_0 \leftarrow$ all receive events of s_0 in sequence;
$receives_1 \leftarrow$ all receive events of s_1 in sequence;
if $sends_0.size! = receives_1.size$ ***Or*** $sends_1.size! = receives_0.size$ **then**
    return False;
for $i \in 0..sends_0.size$ **do**
    if $sends_0[i].payload! = receives_1[i].payload$ **then**
        return False;
for $i \in 0..sends_1.size$ **do**
    if $sends_1[i].payload! = receives_0[i].payload$ **then**
        return False;
return True;

4.6.3 Data Stream Verification Algorithm for Message Queue

Message Queuing 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.[17] In this work, only one sending application versus one receiving application case is considered. Multiple senders to multiple receivers scenario can be divided into multiple sender and receiver situation. Both applications 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 exemplified in Figure4.8.

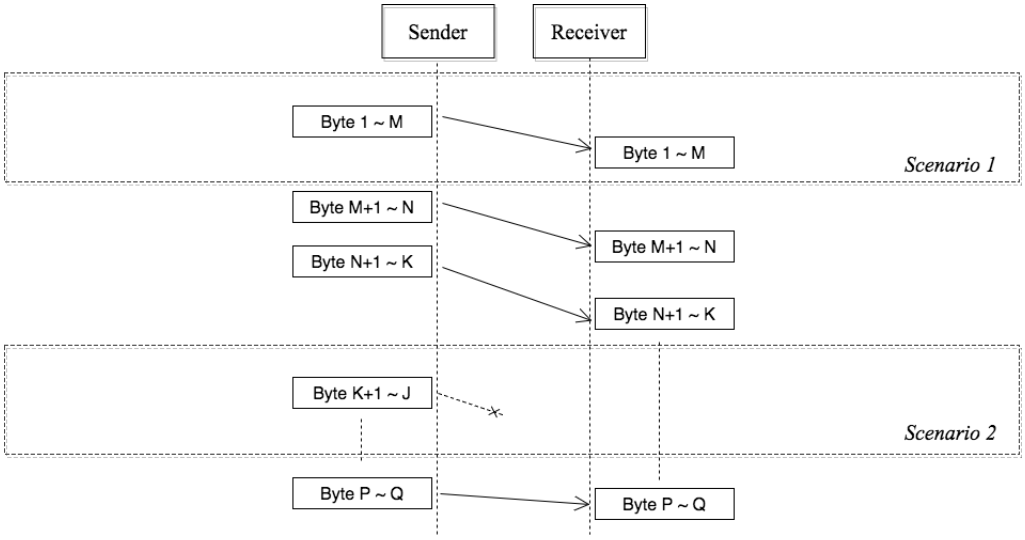

Figure 4.8: Data Transfer Scenarios for Message Queue

To verify the content preservation of the unreliable communication, Algorithm 8 tries to find the match sent packet in the other stream for each received packet. If any of the received packet

can not be match, the content preservation is not satisfied. Since the sent packets are received in order, the searching for each received packet will start from the next index of the last matching sent packet.

Algorithm 8: Data Stream Verification of Message Queue

Input: s_0, s_1

return $satisfied$
$sends_0 \leftarrow$ all sent events of s_0 in sequence;
$sends_1 \leftarrow$ all sent events of s_1 in sequence;
$receives_0 \leftarrow$ all receive events of s_0 in sequence;
$receives_1 \leftarrow$ all receive events of s_1 in sequence;
if $sends_0.size < receives_1.size$ *Or* $sends_1.size < receives_0.size$ **then**
    return False;
$lastMatchIndex = 0$;
for $i \in 0..receives_1.size$ **do**
    $tempIndex = lastMatchIndex$;
    for $j \in lastMatchIndex + 1..sends_0.size$ **do**
        if $sends_0[j].payload = receives_1[i].payload$ **then**
            $lastMatchIndex = j$;
            break the inner For loop;
    `// This received packet can not be matched by any sent packet`
    if $tempIndex = lastMatchIndex$ **then**
        return False;
$lastMatchIndex = 0$;
for $i \in 0..receives_0.size$ **do**
    $tempIndex = lastMatchIndex$;
    for $j \in lastMatchIndex + 1..sends_1.size$ **do**
        if $sends_1[j].payload = receives_0[i].payload$ **then**
            $lastMatchIndex = j$;
            break the inner For loop;
    if $tempIndex = lastMatchIndex$ **then**
        return False;
return True;

The time complexity of this algorithm is $O(N^2 + M^2)$, N and M are the numbers of data sent events of the two streams.

4.6.4 Data Stream Verification Algorithm for 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 exemplified in Figure4.9.

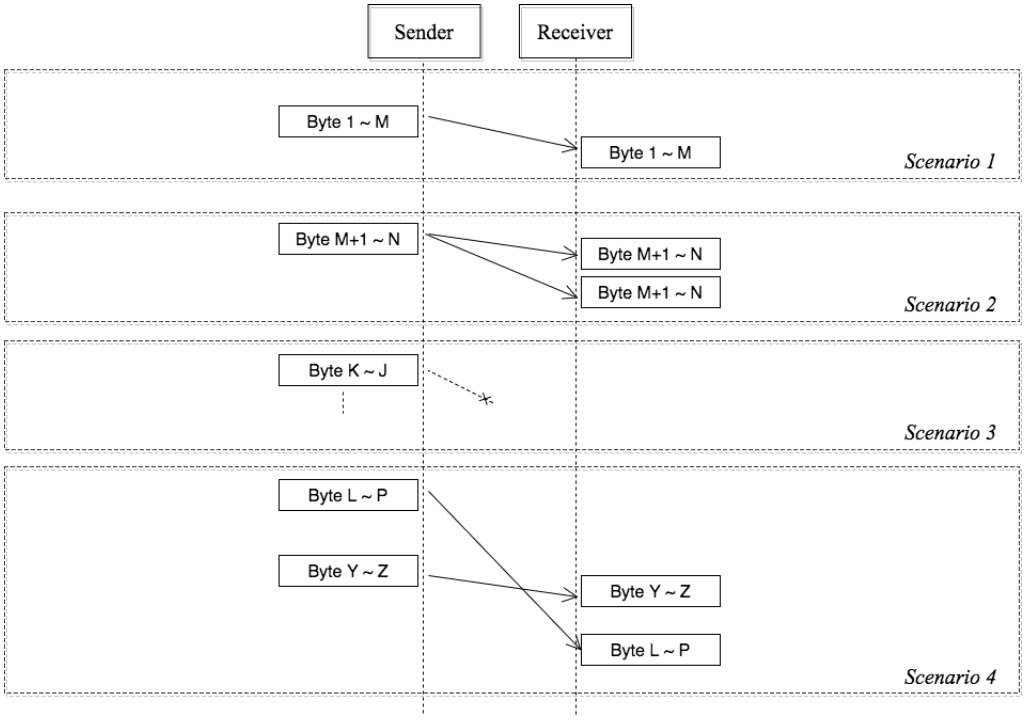

Figure 4.9: Data Transfer Scenarios for UDP

Similar to Message Queue, Algorithm 9 try to find the match sent packet in the other stream for each received packet. If any of the received packet can not be match, the content preservation is not satisfied. However, due to the disordering can happen in UDP, the searching for each received packet will not constraint the searching index. But the matched sent packet will excluded from the following searching, which means each sent packet can only match to one received packet.

Algorithm 9: Transmitted Verification of UDP

Input: s_0, s_1

return $satisfied$
$sends_0 \leftarrow$ all sent events of s_0 in sequence;
$sends_1 \leftarrow$ all sent events of s_1 in sequence;
$receives_0 \leftarrow$ all receive events of s_0 in sequence;
$receives_1 \leftarrow$ all receive events of s_1 in sequence;
if $sends_0.size < receives_1.size$ *Or* $sends_1.size < receives_0.size$ **then**
    return False;
for $i \in 0..receives_1.size$ **do**
    $matchFlag = False$;
    for $j \in 0..sends_0.size$ **do**
        if $sends_0[j].payload = receives_1[i].payload$ **then**
            $matchFlag = True$;
            delete the packet from $sends_0$ break the inner For loop;
        // This received packet can not be matched by any sent packet
    if $matchFlag = False$ **then**
        return False;
for $i \in 0..receives_0.size$ **do**
    $matchFlag = False$;
    for $j \in 0..sends_1.size$ **do**
        if $sends_1[j].payload = receives_0[i].payload$ **then**
            $matchFlag = True$;
            delete the packet from $sends_0$ break the inner For loop;
    if $matchFlag = False$ **then**
        return False;
return True;

The time complexity of this algorithm is $O(N^2 + M^2)$, N and M are the numbers of data sent transfer events in the two streams.

4.7 Communication Identification Process

The general communication identification of the dual_trace consist of the algorithms presented previously in this chapter and can be summarized as the follow steps with the corresponding algorithm for each communication categories(i.e. reliable or unreliable) and each communication methods:

Step 1. Execute the Function Event Reconstruction algorithm to the two traces in the dual_trace

Step 2. Execute the Stream Exaction Algorithm to the both traces

Step 3. Use the Stream Matching Algorithm to match the streams of the two traces

Step 4. Verify the matched streams by the their satisfaction of the content preservation

Step 5. Include the matched streams that satisfy the preservation

Step 6. Retrieve the endpoint $e =< handle, d_r, d_s >$ and channel identifier ch from the matched streams and output a set the communications. (No corresponding algorithm for this step since the retrieval is only reorganize the information in the streams and is trivial)

4.8 Limitation of the Identification

The Identification discussed in this chapter is not perfect. It has two major limitations:

- The timing preservation of the communication is not verified.
- Due to the data transmitted in two communications can be identical(or their difference can not be tell by the content verification), the false negative error of the identification can not be eliminated. Figure4.10 indicates the ineffective and effective identification scenarios.

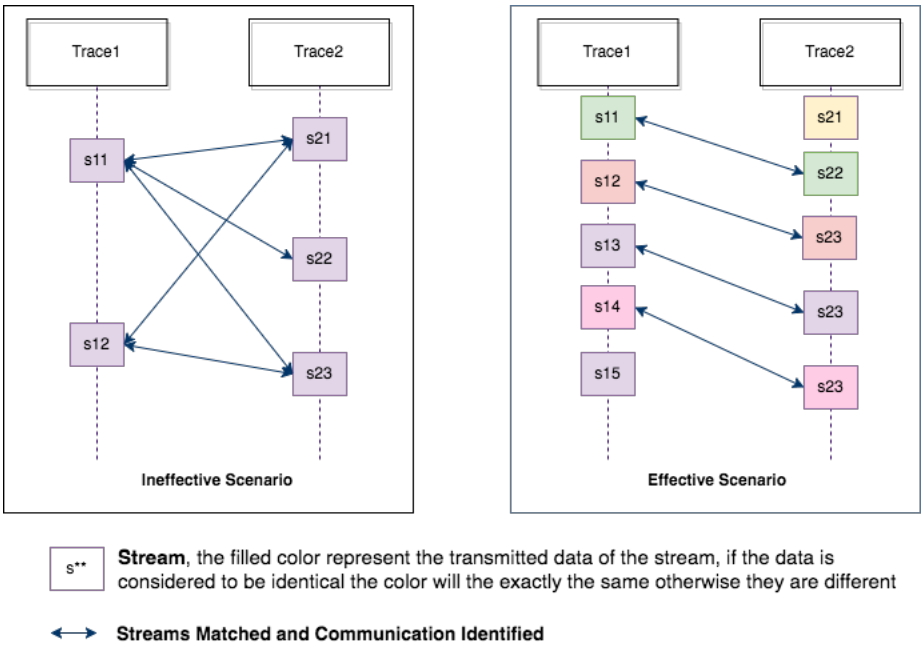

Figure 4.10: Communication Identification Scenarios

Chapter 5

Feature Prototype On Atlantis

In this section, I describe the design of the feature prototype of communication identification from the dual_trace. This feature is implemented on Atlantis and is built on top of Atlantis' other features, such as "memory reconstruction", "function inspect" and "views synchronization". Atlantis is an assembly trace analysis environment. It provides many powerful and novel features to assist assembly level execution trace analysis.[10] This prototype implemented some of the algorithms described in Chapter4 as well as the user interfaces.

This prototype consists of four main components: 1) user defined setting for defining the concerned communication methods' description. 2) a view that can parallelly present both traces in the dual_trace. 3) two identification features: Event Stream identification and communication identification. 4) identification result navigation.

5.1 User Defined Communication Method Description

In Chapter4, I defined the communication method description which is a set of function descriptions of a specific communication method. The communication method descriptions restrict how the communication looks like in the dual_trace and how they can be identified.

However, the communication method description for a communication method can be different depends on the implementation solution of the method. Furthermore, there are so many communication methods in the real world, there is no way that we can define all of them for the users. Instead, a configuration file in Json format is used for the users to define their concerned communication methods and the corresponding description. This setting file will be the input for the communication identification. All concerned communication methods can be listed in this file. The identification features implemented in this prototype iterate all methods in the Json configura-

tion file named “communicationMethods.json” and identify the selected ones. This configuration includes the communication method and their function descriptions. A default template is given for user reference, this default template is generated by Atlantis when it was launched and stored in the .tmp folder in the trace analysis project folder. The users can modify this template as to the communication method of interest. The default template example can be find in SectionB.

5.1.1 Communication Methods' Implementation in Windows

In this section, I present the result of investigation about the implementation of the four communication methods: Named Pipe, Message Queue, TCP and UDP in Windows. By this investigation, the communication descriptions of these methods can be designed.

In the investigation, I reviewed the Windows APIs of the communication methods and their example code.

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 investigated the most basic usage provided in Windows documentation. For each communication method, a system function list is provided for reference as the communication method description. The functions in the description are supported in most Windows operating systems, such as Windows 8, Window 7. The provided function descriptions for a communication method should not be considered as the only combination or solution for that communication method. With the understanding of this, it should be fairly easy to draw out lists for other solutions or other communication methods.

Moreover, the instances of the descriptions only demonstrate Windows C++ APIs. But the idea of the description is generalizable to other operating systems with the effort of understanding the APIs of those operating systems.

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 and register changes of the instructions. The calling convention helps us to understand where the parameters are stored so that we can find them in the memory state of while emulate the execution of the trace. Calling Convention is different for operating systems and the programming language. The Microsoft* x64 example calling convention is listed in Appendix A since we used dual-trace from

Microsoft* x64 for case study in this work.

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.[15] 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. Table5.1 lists the functions of the events for synchronous mode while Table5.2 lists the functions of the events for the asynchronous mode for a Named pipe communication.

Table 5.1: Function Descriptions for Synchronous Named Pipe

Name	**Type**	**Parameter Description**			
		Parameter	**Register**	**In or Out**	**Buffer Or Value**
CreateNamedPipe	open	Handle	RAX:	Out	Value
		FileName	RCX	In	Address
CreateFile	open	Handle	RAX:	Out	Value
		FileName	RCX	In	Address
WriteFile	send	Handle	RCX	In	Value
		SendBuffer	RDX	In	Address
		MessageLength	R9	Out	Value
ReadFile	receive	Handle	RCX	In	Value
		RecvBuffer	RDX	In	Address
		MessageLength	R9	Out	Value
CloseHandle	close	Handle	RCX	In	Value
DisconnectNamedPipe	close	Handle	RCX	In	Value

Table 5.2: Function Descriptions for Asynchronous Named Pipe

Name	**Type**	**Parameter Description**			
		Parameter	**Register**	**In or Out**	**Buffer Or Value**
CreateNamedPipe	open	Handle	RAX:	Out	Value
		FileName	RCX	In	Address
CreateFile	open	Handle	RAX:	Out	Value
		FileName	RCX	In	Address
WriteFile	send	Handle	RCX	In	Value
		SendBuffer	RDX	In	Address
		MessageLength	R9	Out	Value
ReadFile	receive	Handle	RCX	In	Value
		RecvBuffer	RDX	In	Address
		MessageLength	R9	Out	Value
GetOverlappedResult	send/receive	Handle	RCX:	In	Value
		OverlapStruct	RDX	Out	Address
CloseHandle	close	Handle	RCX	In	Value
DisconnectNamedPipe	close	Handle	RCX	In	Value

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. Table5.3 lists the function descriptions for synchronous mode while Table5.4 list the function descriptions for the asynchronous mode without callback. With callback function situation is not elaborated here.

Table 5.3: Function Descriptions for Synchronous Message Queue

Name	**Type**	**Parameter Description**			
		Parameter	**Register**	**In or Out**	**Buffer Or Value**
MQOpenQueue	open	Handle	RAX:	Out	Value
		QueueName	RCX	In	Address
CreateFile	open	Handle	RAX:	Out	Value
		FileName	RCX	In	Address
MQSendMessage	send	Handle	RCX	In	Value
		MessageStruct	RDX	In	Address
MQReceiveMessage	receive	Handle	RCX	In	Value
		MessageStruct	R9	In	Address
MQCloseQueue	close	Handle	RCX	In	Value

Table 5.4: Function Descriptions for Asynchronous Message Queue

Name	**Type**	**Parameter Description**			
		Parameter	**Register**	**In or Out**	**Buffer Or Value**
MQOpenQueue	open	Handle	RAX:	Out	Value
		QueueName	RCX	In	Address
CreateFile	open	Handle	RAX:	Out	Value
		FileName	RCX	In	Address
MQSendMessage	send	Handle	RCX	In	Value
		MessageStruct	RDX	In	Address
MQReceiveMessage	receive	Handle	RCX	In	Value
		MessageStruct	R9	In	Address
MQGetOverlappedResult	receive/send	OverlapStrAndHandle	RCX	In/Out	Address
MQCloseQueue	close	Handle	RCX	In	Value

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 5.5 lists the functions of a UDP or TCP communication.

Table 5.5: Function Descriptions for for TCP and UDP

Name	**Type**	**Parameter Description**			
		Parameter	**Register**	**In or Out**	**Buffer Or Value**
socket	open	Handle	RAX:	Out	Value
bind	open	Handle	RCX:	In	Value
		ServerAddAndPort	RDX	Out	Address
accept	open	ChildHandle	RAX	Out	Vaule
		Handle	RCX	In	Value
		ClientAddAndPort	RDX	In	Value
send	send	Handle	RCX:	In	Value
		SendBuffer	RCX	In	Address
recv	receive	Handle	RAX	In	Value
		RecvBuffer	RCX	Out	Address
closesocket	close	Handle	RCX	In	Value

5.2 Parallel Editor View For Dual_Trace

The dual_trace consist of two execution traces which are interacting with each other. Presenting them in the same view makes the analysis for the user much easier. To open parallel editor view, the user need to open one trace as the normal one and the other as the dual_trace of the opened one. A new menu option in the project navigation view are added to open the second trace as the dual_trace of the active trace. The implementation of the parallel editor take the advantage of the existing SWT of Eclipse plug-in development. The detail of the implementation can be found in Appendix C. Figure 5.1 shows this menu option and Figure C shows the parallel editor view.

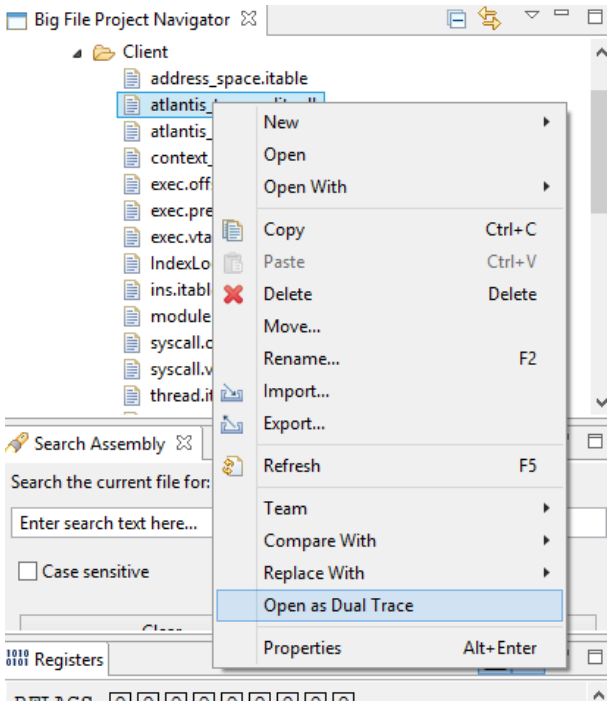

Figure 5.1: Menu Item for opening Dual_trace

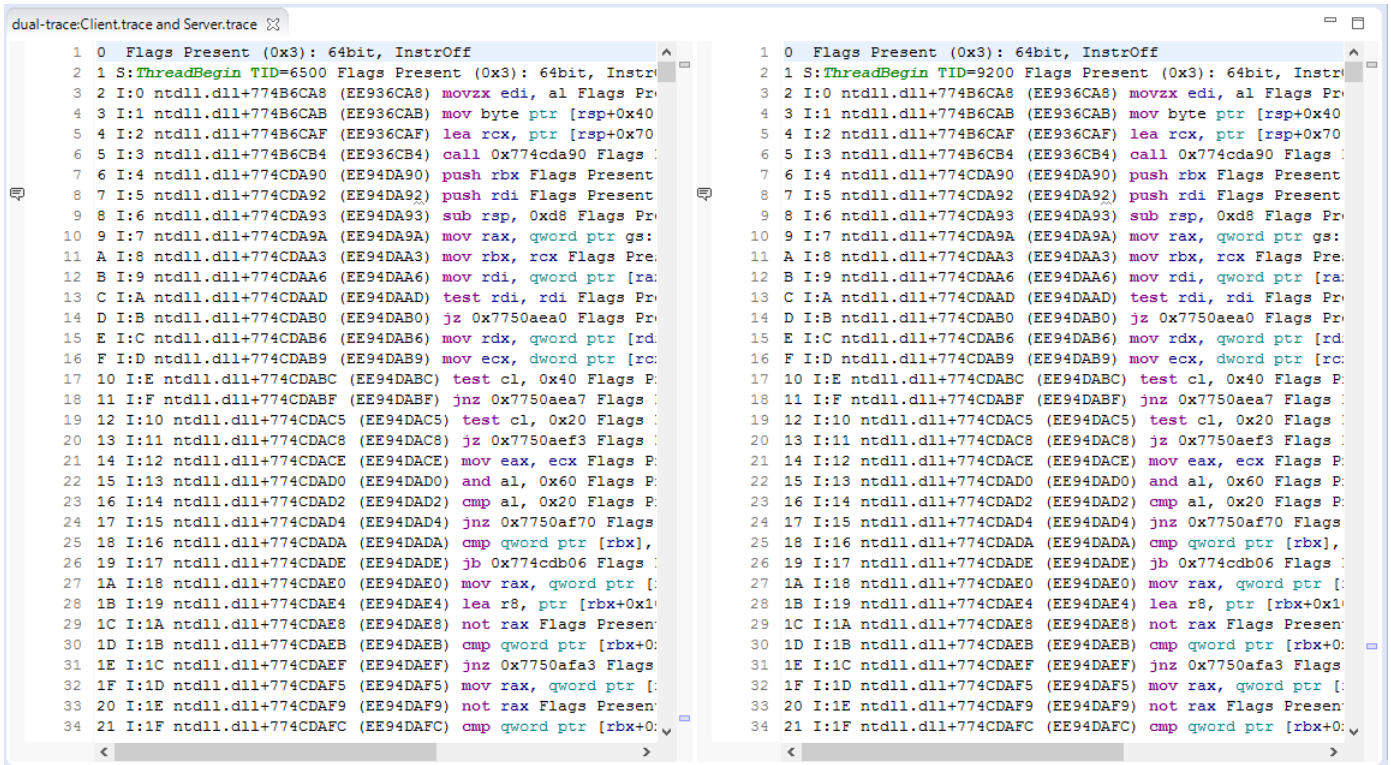

Figure 5.2: Parallel Editor View

5.3 Identification Features

I implemented two identification features, one is stream identification for both traces in the dual_trace, the other is the communication identification. These two features implemented the "event stream extraction algorithm" and "event stream matching algorithm" designed in Chapter4. The implementation of these two identification features relies on the existing "function inspect" feature of Atlantis. 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. By importing the DLLs and executable binary, Atlantis can recognize the function call from the execution trace by the function names. Therefore the corresponding Dlls or executable binaries for both traces in the dual_trace have to be loaded into Atlantis before conducting the identification.

A new menu "Dual_trace Tool" with three menu options is designed for these two identification features. In this menu, two options are for conducting the identification which are "Stream Identification" and "Communication Identification" while one is for loading the DLLs and executable binary which is "Load Library Exports". Currently, the "Load library export" function can only load libraries for the trace in the active editor. So this item in the menu has to be run twice separately for each trace of the dual_trace. Figure5.3 shows this new menu in Atlantis. When the user perform any of the identification features, there is the prompt dialog as shown in Figure 5.4 which asks the user what communication methods they want to identify from the dual_trace. This list is provided by the configuration file I mention in Section 5.1. The user can select one or multiple methods.

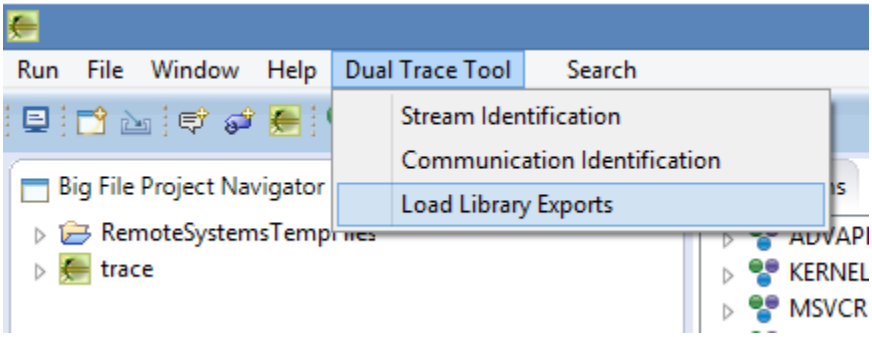

Figure 5.3: Dual_trace Tool Menu

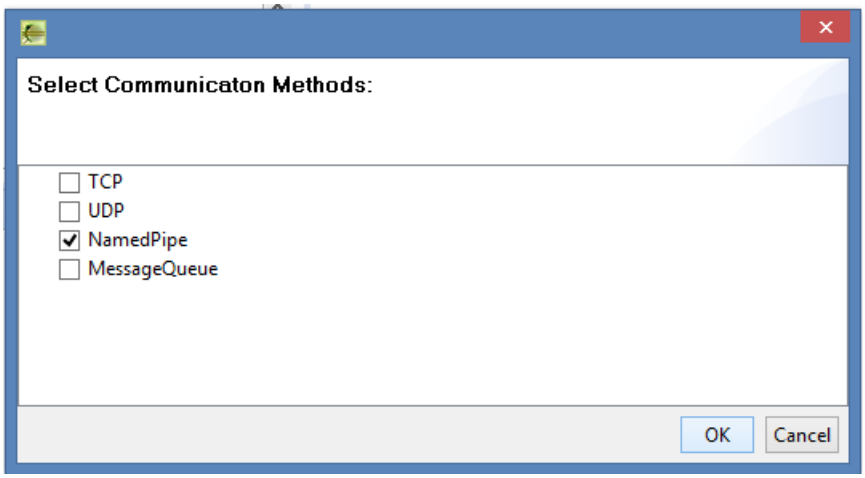

Figure 5.4: Prompt Dialog for Communication Selection

A new view named “Communication” is designed for presenting the result of the identification of streams and communications. Since the user can have multiple selection for communication methods they concern, the output identification result contains all the identified communications or streams of all the concerned communication methods and the identified results are clustered by methods. There are two sub tables in this view, the left one is for the stream identification result while the left one is for communication identification result. The reason for putting this two result in the same view is for easy access and comparison of the data for the users. Figure 5.5 shows this view with result data in it. Each time when the user rerun the identification features the result in the corresponding table will be refreshed to show only the latest identification result. But the other table will not be affected. For example, if the user run the “Stream Identification” feature first, the stream identification result will show on the left table of the view. And then the user run the “communication Identification”, the communication identification result will be shown on the right table while the left one still holding the last stream identification result.

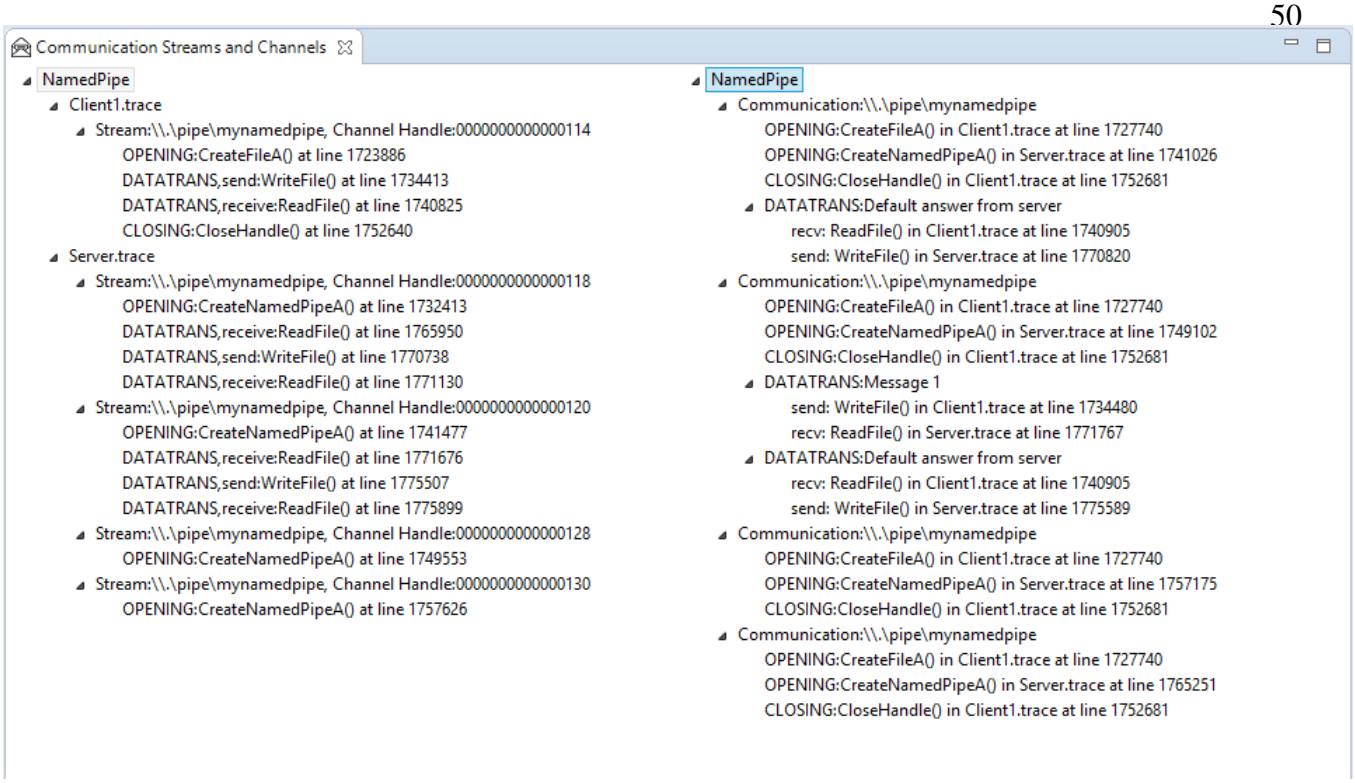

Figure 5.5: Communication View for Showing Identification Result

5.4 Identification Result View and Result Navigation

Atlantis is a analysis environment that has various views to allow user access to different information from the trace, such as the memory and register state of the current instruction line. Moreover, these views synchronize automatically with the editor view. These functionality and information also benefit the communication analysis of the dual_trace. Providing the user a way to navigate from the identified result to the traces in the editors allows them to take advantage of the current existing functionality of Atlantis and make their analysis of the dual_trace more efficient.

In the result list, each event entry is corresponding to a function call. The functions were called at function call line and all the inputs of the function calls can be recovered from the memory state of this instruction line. The functions returned at the return instruction lines, all the outputs of the function calls can be recovered in the memory state of the the return instruction line. From the event entries, this implementation provide two different ways for the user to navigate back to where the function begins and ends. When the user “double click” on an entry, it will bring the user to the start line of the function in the corresponding trace editor. When the the right click on the event entry, a prompted menu with the option “Go To Line of Function End” will show up as

in Figure 5.6. Clicking on this option will bring the user to the return line of this function in the trace editor. All other views update immediately with this navigation.

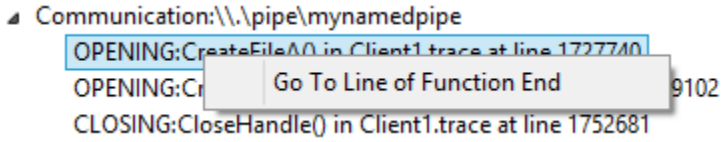

Figure 5.6: Right Click Menu on Event Entry

Moreover, the “remove” option as shown in Figure 5.7 in the right click menu on the “stream or “communication” entries is provided for the user to remove the selected “stream” or “communication” entry. This provides the user the flexibility to get rid of the data that they don’t care.

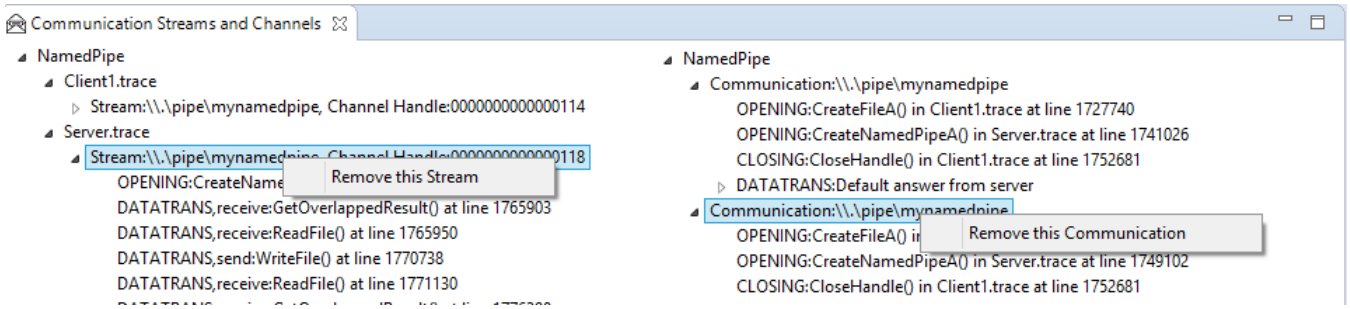

Figure 5.7: Right Click Menu on Event Entry

Chapter 6

Proof of Concept

In this section, I present two experiments I ran for the proof of concept of the communication analysis though execution traces.

These experiments aimed to test the communication model and the communication identification approach. They also verify the design of the some algorithms, for their correctness.

User case study is not included in this thesis and can be the future work. The feature prototype implementation is not evaluated and can be part of the user case study. But I used the implemented feature on Atlantis to conduct the experiments.

I first present the design of the experiments and their result. And then, I discuss the result of the experiments.

6.1 Experiments

In this section, I describe the design of the experiments. Two experiments are conducted in this research. All test programs in these two experiments were written in C++ and the source code can be found in Appendix D. Our research partner DRDC executed the programs in their environment and provided the captured traces, the used .dll files and the source code of the programs for the experiments.

Results are provided for each experiment. Both of the conducted experiments were about named pipe communication method. The following two subsections provides the details of the experiments and their result.

6.1.1 Experiment 1

In the first experiment, two programs communicated with each other through a synchronous Named pipe channel. One of the programs acted as the Named pipe server while the other as the client. Figure 6.1 is the sequence diagram of the interaction between the server and client. Traces were captured while these two program were running and interacting. The two captured traces were analysed as dual_trace $exp1$ in this experiment. I used the implemented features in Atlantis to analyse this dual_trace. I ran the "Stream identification" and "Communication identification" operations for this dual_trace. The identified streams, communication and the processing time are showed in Figure6.2.

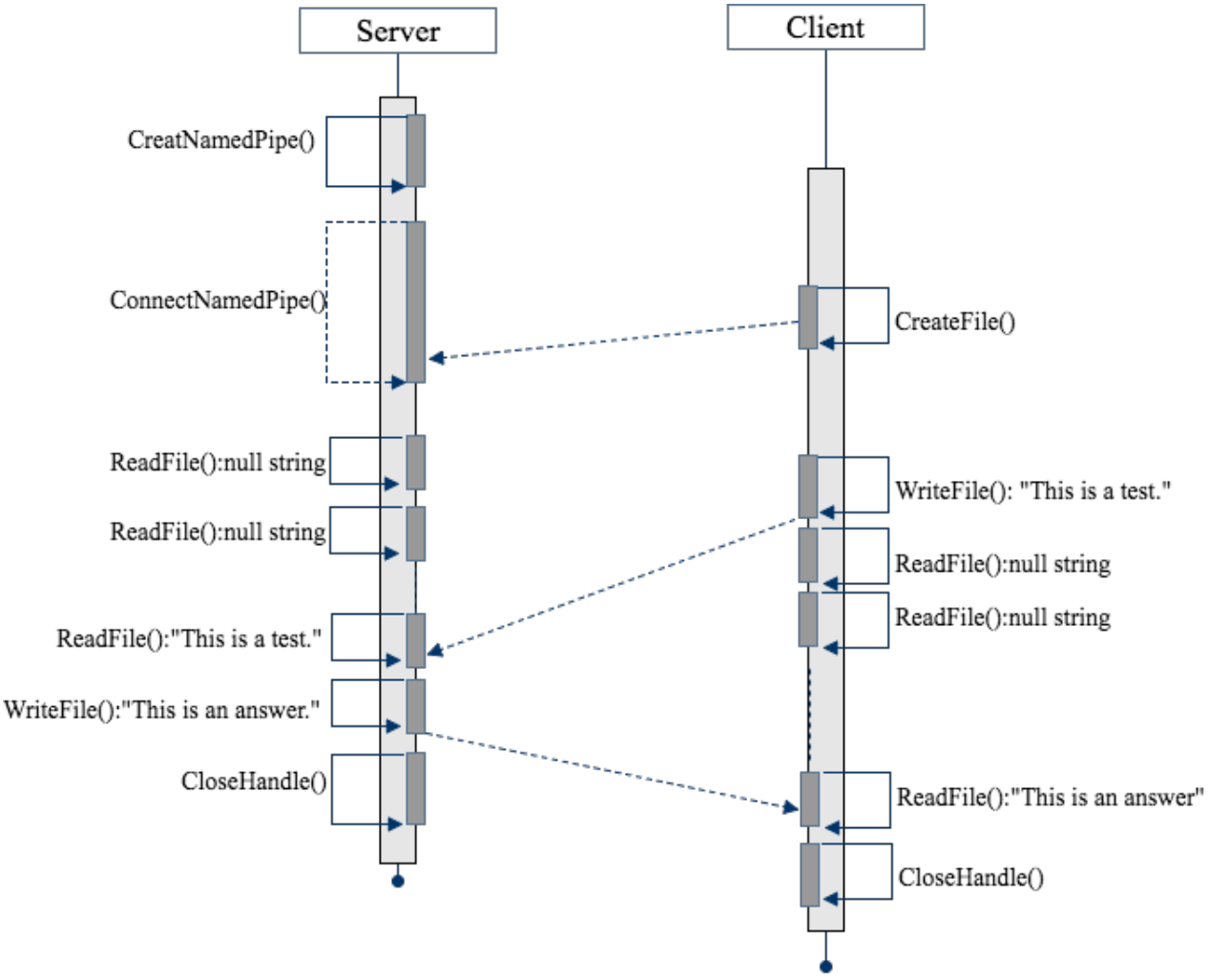

Figure 6.1: Sequence Diagram of Experiment 1

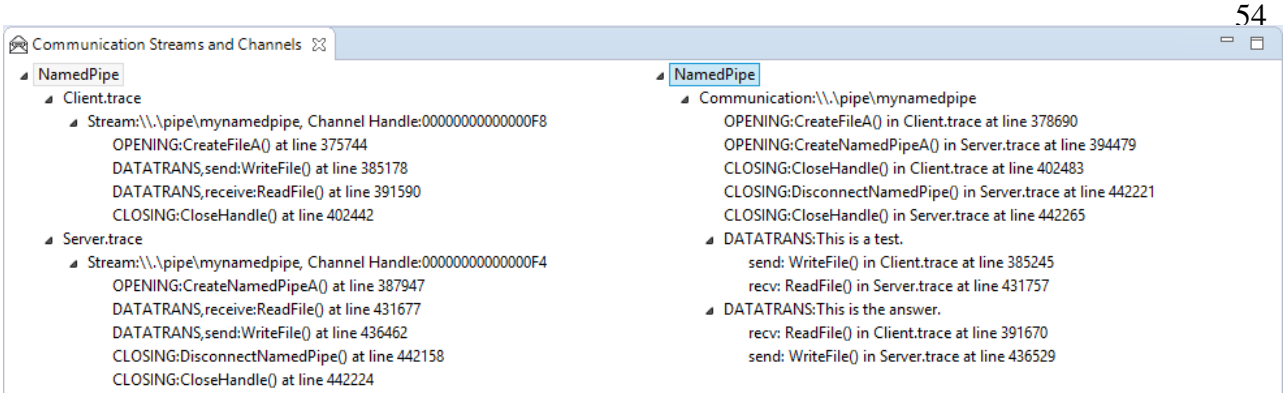

Figure 6.2: Identification result of $exp1$

6.1.2 Experiment 2

In the second experiment, one program was running as the Named pipe server. In this server program, four named pipes were created and can be connected by up to four client at a time. Two other programs as the Named pipe clients connected to this server. Those two clients (client 1 and client 2) used the identical program but run in sequence. Figure 6.3 is the sequence diagram of the interaction among the server and clients. The function calls' sequence is only a possible combination from analyzing the source code. The real happening sequence can varies. Traces were captured at the time when these three programs were running and interacting. One trace for each program. The three traces are analysed as two dual_traces, $exp2.1$ and $exp2.2$. $exp2.1$ consists of traces of server and client 1 and $exp2.2$ consists of traces of server and client 2. I ran the "Stream identification" and "Communication identification" operations for these two dual_trace. The identified streams, communications are shown in Figure 6.4 and Figure 6.5.

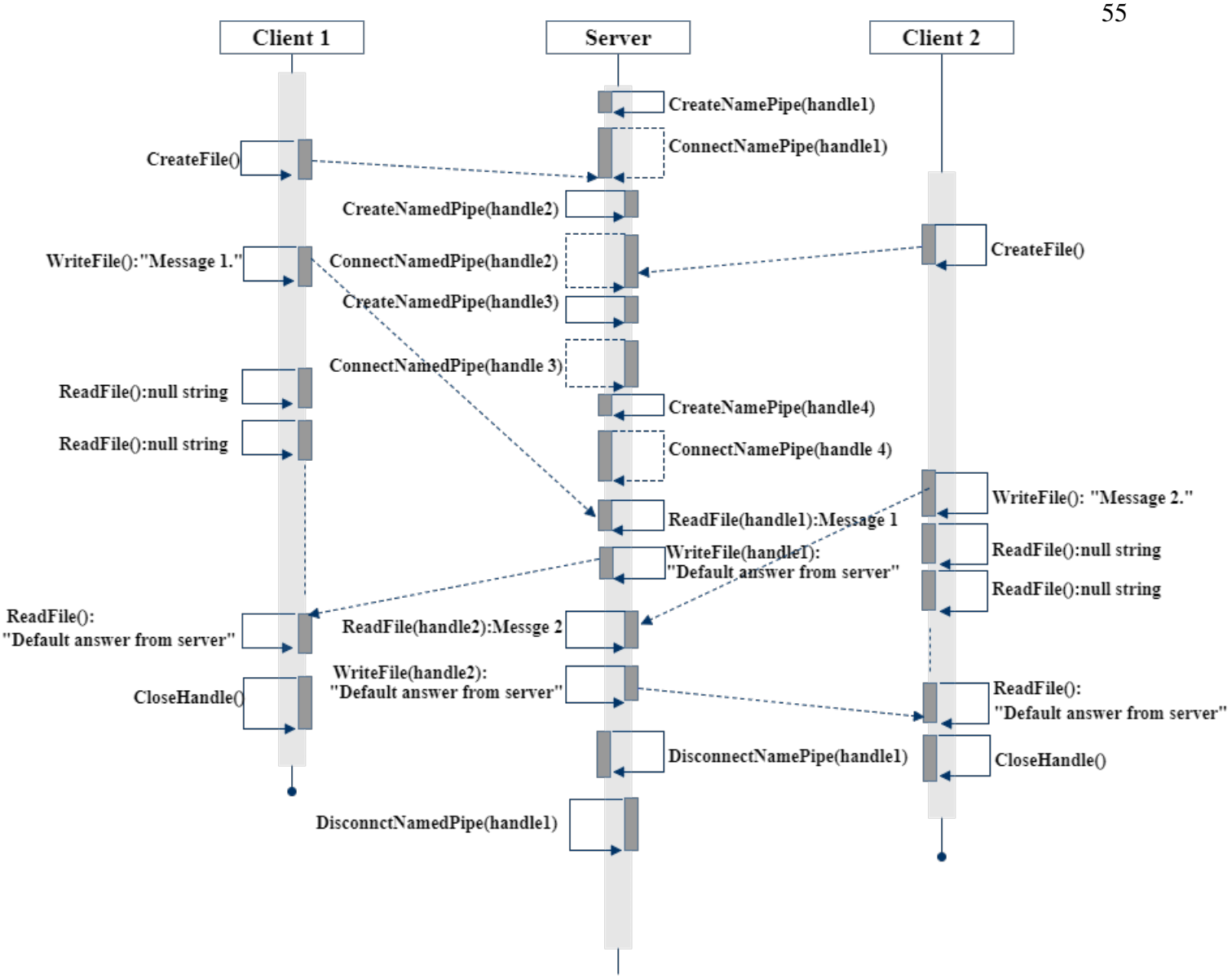

Figure 6.3: Sequence Diagram of Experiment 2

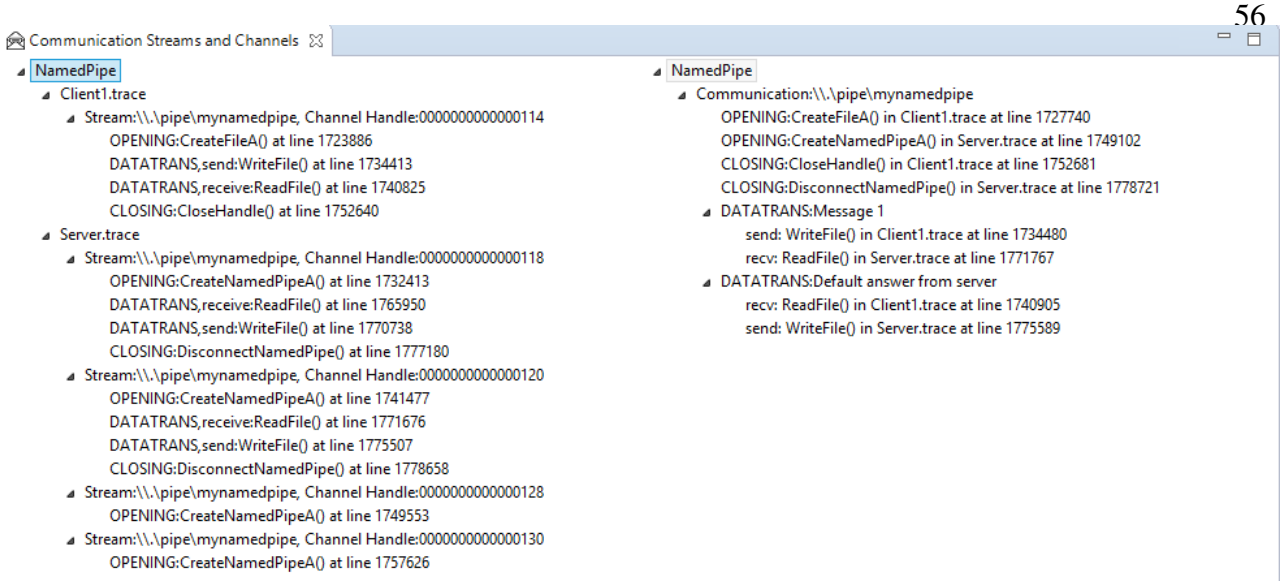

Figure 6.4: Identification result of $exp2.1$

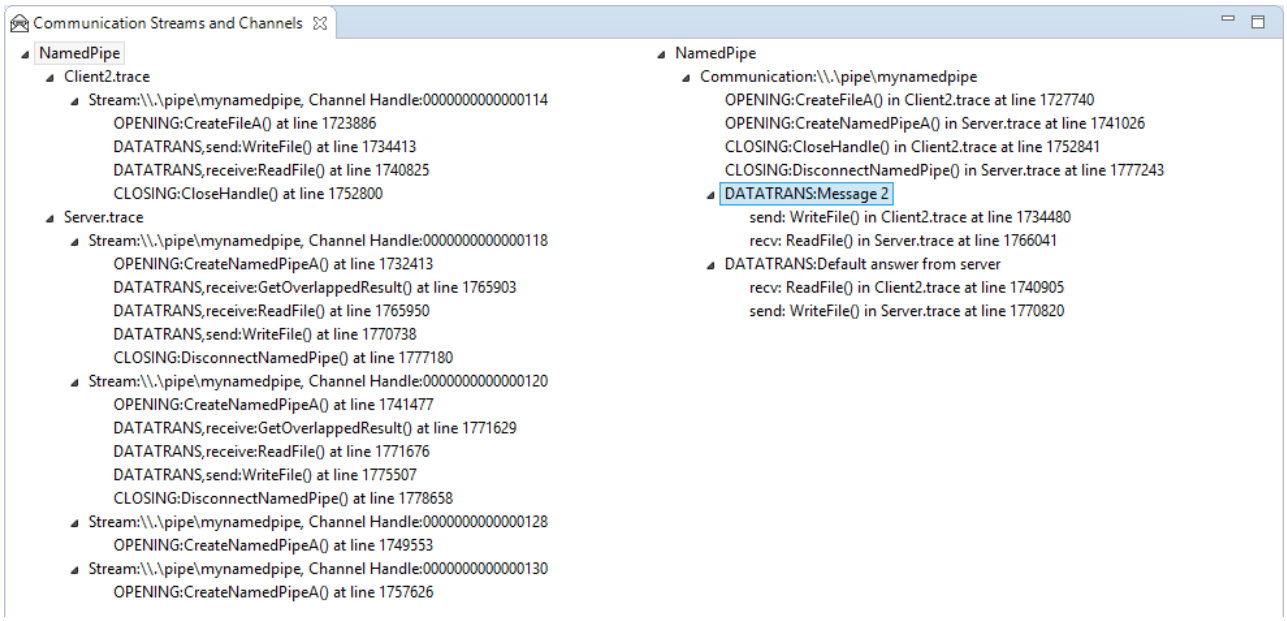

Figure 6.5: Identification result of $exp2.1$

6.2 Discussion

In the result of $exp1$, there are one stream identified in client trace and one in server trace, and these two streams are matched into a communication of this dual_trace. This identification result

represents the actual communication happen between the named pipe server and client. In the result of $exp2.1$ and $exp2.2$, there are one stream identified in client traces and four in server trace respectively for each dual_trace. The streams are further matched and verified and eventually one communication is identified for each dual_trace. The result aligns to the sequence diagram in Figure6.3.

Chapter 7

Conclusions and Future Work

In this thesis, I present the approach of communication analysis through assembly level execution traces. I first defined the communication model. This abstract model depicts the outline of a communication between two running program, which gives the ground rule for the communication analysis. Then I depicted the general definition of the dual_trace, the definition indicate that all traces comply to this definition can be used to conducted the communication analysis.

I also developed the essential algorithms for the communication identification. The high level algorithm is generalizable for all communication methods' identification while the event stream extraction and matching algorithm are distinct for each communication method according to their channel open and data transfer mechanisms. However, the developed algorithms provides clear and referable examples to develop your own algorithm for communication methods which are not discussed in this thesis.

On top of the existing execution trace analysis environment Atlantis, I implemented the communication identification features. This feature provides the users a way to extend their concerned communication methods through the configuration file. The user interface allows the users to conduct the communication and stream identification from the dual_traces and navigate back from the identified result to the views of the trace in Atlantis. This feature prototype is a novel feature for conducting multiple trace analysis for reverse engineering at the time when this thesis was written. The experiments conducted in this work preliminary proves the usability of the model and the algorithms.

This thesis illustrates the novel idea and approach for dynamic program analysis which considerate the interaction of two programs. This idea is valuable due to the fact that programs or malware in the real world work collaboratively. The analysis of the communication and interaction of the programs provide more reliable information for vulnerability detection and program

analysis.

Future work can be divided in two directions. One is extending the model to be more generalize for all kinds of interaction but not only the message transferring communications while the other is conducting user studies of the communication analysis approach and the feature design to get a more concrete result of their usefulness.

Bibliography

[1] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*, pages 5–23. Springer, 2004.

[2] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163. ACM, 2006.

[3] Derek Bruening. Qz: Dynamorio: Dynamic instrumentation tool platform.

[4] Jun Cai, Peng Zou, Jinxin Ma, and Jun He. Sworddta: A dynamic taint analysis tool for software vulnerability detection. *Wuhan University Journal of Natural Sciences*, 21(1):10–20, 2016.

[5] Gerald Combs. Wireshark Go Deep.

[6] 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.

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

[8] José M Garrido. Inter-process communication. *Performance Modeling of Operating Systems Using Object-Oriented Simulation: A Practical Introduction*, pages 169–189, 2000.

[9] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[10] 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.

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

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

[13] Mujtaba Khambatti-Mujtaba. Named pipes, sockets and other ipc.

[14] Hongzhe Li, Taebeom Kim, Munkhbayar Bat-Erdene, and Heejo Lee. Software vulnerability detection using backward trace analysis and symbolic execution. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 446–454. IEEE, 2013.

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

[16] Christopher R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Phys. Rev. E*, 68:046116, Oct 2003.

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

[18] Andreas Sailer, Michael Deubzer, Gerald Lüttgen, and Jürgen Mottok. Coretana: A trace analyzer for reverse engineering real-time software. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 657–660. IEEE, 2016.

[19] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504. ACM, 2016.

[20] tcpdump. Tcpdump/Libpcap public repository.

[21] Jonas Trümper, Stefan Voigt, and Jürgen Döllner. Maintenance of embedded systems: Supporting program comprehension using dynamic analysis. In *Software Engineering for Embedded Systems (SEES), 2012 2nd International Workshop on*, pages 58–64. IEEE, 2012.

[22] Shameng Wen, Qingkun Meng, Chao Feng, and Chaojing Tang. A model-guided symbolic execution approach for network protocol implementations and vulnerability detection. *PloS one*, 12(11):e0188229, 2017.

[23] Oleh Yuschuk. Ollydbg, 2007.

[24] Dazhi Zhang, Donggang Liu, Yu Lei, David Kung, Christoph Csallner, and Wenhua Wang. Detecting vulnerabilities in c programs using trace-based testing. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 241–250. IEEE, 2010.

Appendix A

Microsoft x64 Calling Convention for C/C++

- RCX, RDX, R8, R9 are used for integer and pointer arguments in that order left to right.
- XMM0, 1, 2, and 3 are used for floating point arguments.
- Additional arguments are pushed on the stack left to right. ...
- Parameters less than 64 bits long are not zero extended; the high bits contain garbage.
- Integer return values (similar to x86) are returned in RAX if 64 bits or less.
- Floating point return values are returned in XMM0.
- 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.

Appendix B

Function Set Configuration Example

Listing B.1: communicationMethods.json

```
[
  {
    "communicationMethod": "NamedPipe",
    "funcList": [
      {
        "retrunValReg": {
          "name": "RAX",
          "valueOrAddress": true
        },
        "valueInputReg": {
          "name": "RCX",
          "valueOrAddress": false
        },
        "functionName": "CreateNamedPipeA",
        "createHandle": true,
        "type": "open"
      },
            {
        "retrunValReg": {
          "name": "RAX",
          "valueOrAddress": true
        },
        "valueInputReg": {
          "name": "RCX",
          "valueOrAddress": false
        },
        "functionName": "ConnectNamedPipe",
        "createHandle": false,
        "type": "open"
      },
      {
        "retrunValReg": {
          "name": "RAX",
```

```
      "valueOrAddress": true
    },
    "valueInputReg": {
      "name": "RCX",
      "valueOrAddress": false
    },
    "functionName": "CreateFileA",
    "createHandle": true,
    "type": "open"
  },
       {
    "retrunValReg": {
      "name": "RAX",
      "valueOrAddress": value
    },
    "valueInputReg": {
      "name": "RCX",
      "valueOrAddress": value
    },
    "memoryInputReg": {
      "name": "RDX",
      "valueOrAddress": address
    },
    "memoryInputLenReg": {
      "name": "R8",
      "valueOrAddress": value
    },
    "functionName": "WriteFile",
    "createHandle": false,
    "type": "send"
  },
  {
    "retrunValReg": {
      "name": "RAX",
      "valueOrAddress": value
    },
    "valueInputReg": {
      "name": "RCX",
      "valueOrAddress": value
    },
    "memoryOutputReg": {
      "name": "RDX",
      "valueOrAddress": address
    },
    "memoryOutputBufLenReg": {
      "name": "R8",
      "valueOrAddress": value
    },
    "functionName": "ReadFile",
    "createHandle": false,
    "type": "recv",
    "outputDataAddressIndex": "NamedPipeChannelRDX"
```

```
      },
      {
        "retrunValReg": {
          "name": "RAX",
          "valueOrAddress": value
        },
        "valueInputReg": {
          "name": "RCX",
          "valueOrAddress": value
        },
        "memoryOutputReg": {
          "name": "RDX",
          "valueOrAddress": address
        },
        "functionName": "GetOverlappedResult",
        "createHandle": false,
        "type": "check",
        "outputDataAddressIndex": "NamedPipeChannelRDX"
      },
            {
        "retrunValReg": {
          "name": "RAX",
          "valueOrAddress": value
        },
        "valueInputReg": {
          "name": "RCX",
          "valueOrAddress": value
        },
        "functionName": "CloseHandle",
        "createHandle": false,
        "type": "close"
      }
            {
        "retrunValReg": {
          "name": "RAX",
          "valueOrAddress": value
        },
        "valueInputReg": {
          "name": "RCX",
          "valueOrAddress": value
        },
        "functionName": "DisconnectNamedPipe",
        "createHandle": false,
        "type": "close"
      }
    ]
  }
]
```

Appendix C

Code of the Parallel Editors

Two essential pieces of code are listed for the parallel editor. One is for splitting the editor area for two editors while the other is to get the active parallel editors later on for dual_trace analysis.

C.1 The Editor Area Split Handler

Listing C.1: code in OpenDualEditorsHandler.java

```
public class OpenDualEditorsHandler extends AbstractHandler {
      EModelService ms;
      EPartService ps;
      WorkbenchPage page;


   public Object execute(ExecutionEvent event) throws ExecutionException {
            IEditorPart editorPart = HandlerUtil.getActiveEditor(event);
            if (editorPart == null) {
                  Throwable throwable = new Throwable("No active editor");
                  BigFileApplication.showErrorDialog("No active editor", "Please open one file
                      ↪ first", throwable);
                  return null;
            }

            MPart container = (MPart) editorPart.getSite().getService(MPart.class);
            MElementContainer m = container.getParent();
            if (m instanceof PartSashContainerImpl) {
                  Throwable throwable = new Throwable("The active file is already opened in one
                      ↪ of the parallel editors");
                  BigFileApplication.showErrorDialog("TThe active file is already opened in one
                      ↪ of the parallel editors",
                              "The active file is already opened in one of the parallel editors",
                                  ↪  throwable);
                  return null;
```

```
            }
            IFile file = getPathOfSelectedFile(event);

            IEditorDescriptor desc = PlatformUI.getWorkbench().getEditorRegistry().
                ↪ getDefaultEditor(file.getName());
            try {
                  IFileUtils fileUtil = RegistryUtils.getFileUtils();
                  File f = BfvFileUtils.convertFileIFile(file);
                  f = fileUtil.convertFileToBlankFile(f);
                  IFile convertedFile = ResourcesPlugin.getWorkspace().getRoot().
                      ↪ getFileForLocation(Path.fromOSString(f.getAbsolutePath()));
                  convertedFile.getProject().refreshLocal(IResource.DEPTH_INFINITE, null);
                  if (!convertedFile.exists()) {
                        createEmptyFile(convertedFile);
                  }

                  IEditorPart containerEditor = HandlerUtil.getActiveEditorChecked(event);
                  IWorkbenchWindow window = HandlerUtil.getActiveWorkbenchWindowChecked(event);
                  ms = window.getService(EModelService.class);
                  ps = window.getService(EPartService.class);
                  page = (WorkbenchPage) window.getActivePage();
                  IEditorPart editorToInsert = page.openEditor(new FileEditorInput(convertedFile)
                      ↪ , desc.getId());
                  splitEditor(0.5f, 3, editorToInsert, containerEditor, new FileEditorInput(
                      ↪ convertedFile));
                  window.getShell().layout(true, true);


            } catch (CoreException e) {
                  e.printStackTrace();
            }

            return null;
      }

   private void createEmptyFile(IFile file) {
            byte[] emptyBytes = "".getBytes();
            InputStream source = new ByteArrayInputStream(emptyBytes);
            try {
                  createParentFolders(file);
                  if(!file.exists()){
                        file.create(source, false, null);
                  }
            } catch (CoreException e) {
                  e.printStackTrace();
            }finally{
                  try {
                        source.close();
                  } catch (IOException e) {
                        // Don't care
                  }
            }
```

```
}

private void splitEditor(float ratio, int where, IEditorPart editorToInsert, IEditorPart
    ↪ containerEditor,
          FileEditorInput newEditorInput) {
      MPart container = (MPart) containerEditor.getSite().getService(MPart.class);
      if (container == null) {
            return;
      }

      MPart toInsert = (MPart) editorToInsert.getSite().getService(MPart.class);
      if (toInsert == null) {
            return;
      }

      MPartStack stackContainer = getStackFor(container);
      MElementContainer<MUIElement> parent = container.getParent();
      int index = parent.getChildren().indexOf(container);
      MStackElement stackSelElement = stackContainer.getChildren().get(index);

      MPartSashContainer psc = ms.createModelElement(MPartSashContainer.class);
      psc.setHorizontal(true);
      psc.getChildren().add((MPartSashContainerElement) stackSelElement);
      psc.getChildren().add(toInsert);
      psc.setSelectedElement((MPartSashContainerElement) stackSelElement);

      MCompositePart compPart = ms.createModelElement(MCompositePart.class);
      compPart.getTags().add(EPartService.REMOVE_ON_HIDE_TAG);
      compPart.setCloseable(true);
      compPart.getChildren().add(psc);
      compPart.setSelectedElement(psc);
      compPart.setLabel("dual-trace:" + containerEditor.getTitle() + " and " +
          ↪ editorToInsert.getTitle());

      parent.getChildren().add(index, compPart);
      ps.activate(compPart);

}

private MPartStack getStackFor(MPart part) {
      MUIElement presentationElement = part.getCurSharedRef() == null ? part : part.
          ↪ getCurSharedRef();
      MUIElement parent = presentationElement.getParent();
      while (parent != null && !(parent instanceof MPartStack))
            parent = parent.getParent();

      return (MPartStack) parent;
}


private IFile getPathOfSelectedFile(ExecutionEvent event) {
      IWorkbenchWindow window = PlatformUI.getWorkbench().getActiveWorkbenchWindow();
```

```
            if (window != null) {
                  window = HandlerUtil.getActiveWorkbenchWindow(event);
                  IStructuredSelection selection = (IStructuredSelection) window.
                      ↪ getSelectionService().getSelection();
                  Object firstElement = selection.getFirstElement();
                  if (firstElement instanceof IFile) {
                        return (IFile) firstElement;
                  }
                  if (firstElement instanceof IFolder) {
                        IFolder folder = (IFolder) firstElement;
                        AtlantisBinaryFormat binaryFormat = new AtlantisBinaryFormat(
                                    folder.getRawLocation().makeAbsolute().toFile());
                        // arbitrary, just any file in the binary set is needed
                        return AtlantisFileUtils.convertFileIFile(binaryFormat.getExecVtableFile
                            ↪ ());
                  }
            }
            return null;
      }
}
```

C.2 Get the Active Parallel Editors

Listing C.2: code for getting parallel editors

```
IEditorPart editorPart = PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage().
    ↪ getActiveEditor();
            MPart container = (MPart) editorPart.getSite().getService(MPart.class);
            MElementContainer m = container.getParent();
            if (!(m instanceof PartSashContainerImpl)) {
                  Throwable throwable = new Throwable("This is not a dual-trace");
                  BigFileApplication.showErrorDialog("This is not a dual-trace!", "Open a dual-
                      ↪ trace First", throwable);
                  return;
            }

            MPart editorPart1 = (MPart) m.getChildren().get(0);
            MPart editorPart2 = (MPart) m.getChildren().get(1);
```

Appendix D

Code of the Programs in the Experiments

D.1 Experiment 1

The two interacting programs were Named pipe server and client. The first piece of code listed below is the code for the server's program while the second piece is for the client program.

Listing D.1: NamedPipeServer.cpp

```
// Example code from: https://msdn.microsoft.com/en-us/library/windows/desktop/aa365588(v=vs.85).
    ↪ aspx

#include <Windows.h>
#include <stdio.h>
#include <strsafe.h>

#define BUFSIZE 512

DWORD WINAPI InstanceThread(LPVOID);
VOID GetAnswerToRequest(char *, char *, LPDWORD);

int main(VOID) {
      BOOL fConnected = FALSE;
      DWORD dwThreadId = 0;
      HANDLE hPipe = INVALID_HANDLE_VALUE, hThread = NULL;
      char *lpszPipename = "\\\\.\\pipe\\mynamedpipe";

      // The main loop creates an instance of the named pipe and
      // then waits for a client to connect to it. When the client
      // connects, a thread is created to handle communications
      // with that client, and this loop is free to wait for the
      // next client connect request. It is an infinite loop.
      for (;;) {
            hPipe = CreateNamedPipe(
                  lpszPipename,    // pipe name
                  PIPE_ACCESS_DUPLEX,   // read/write access
```

```
                PIPE_TYPE_MESSAGE |   // message type pipe
                PIPE_READMODE_MESSAGE |  // message-read mode
                PIPE_WAIT,     // blocking mode
                PIPE_UNLIMITED_INSTANCES, // max. instances
                BUFSIZE,     // output buffer size
                BUFSIZE,     // input buffer size
                0,       // client time-out
                NULL);      // default security attribute

          if (hPipe == INVALID_HANDLE_VALUE) {
                return -1;
          }

          // Wait for the client to connect; if it succeeds,
          // the function returns a nonzero value. If the function
          // returns zero, GetLastError returns ERROR_PIPE_CONNECTED.
          fConnected = ConnectNamedPipe(hPipe, NULL) ? TRUE : (GetLastError() ==
              ↪ ERROR_PIPE_CONNECTED);

          if (fConnected) {
                // Create a thread for this client
                hThread = CreateThread(
                      NULL,    // no security attribute
                      0,     // default stack size
                      InstanceThread,  // thread proc
                      (LPVOID)hPipe,  // thread parameter
                      0,     // not suspended
                      &dwThreadId);  // returns thread ID

                if (hThread == NULL) {
                      return -1;
                }
                else CloseHandle(hThread);
          }
          else
                // The client could not connect, so close the pipe.
                CloseHandle(hPipe);
      }
      return 0;
}

// This routine is a thread processing function to read from and reply to a client
// via the open pipe connection passed from the main loop. Note this allows
// the main loop to continue executing, potentially creating more theads of
// this procedure to run concurrently, depending on the number of incoming
// client connections.
DWORD WINAPI InstanceThread(LPVOID lpvParam) {
      HANDLE hHeap = GetProcessHeap();
      char *pchRequest = (char *)HeapAlloc(hHeap, 0, BUFSIZE);
      char *pchReply = (char *)HeapAlloc(hHeap, 0, BUFSIZE);

      DWORD cbBytesRead = 0, cbReplyBytes = 0, cbWritten = 0;
```

```
BOOL fSuccess = FALSE;
HANDLE hPipe = NULL;

// Do some extra error checking since the app will keep running even if this
// thread fails.
if (lpvParam == NULL) {
      if (pchReply != NULL) HeapFree(hHeap, 0, pchReply);
      if (pchRequest != NULL) HeapFree(hHeap, 0, pchRequest);
      return (DWORD)-1;
}

if (pchRequest == NULL) {
      if (pchReply != NULL) HeapFree(hHeap, 0, pchReply);
      return (DWORD)-1;
}

if (pchReply == NULL) {
      if (pchRequest != NULL) HeapFree(hHeap, 0, pchRequest);
      return (DWORD)-1;
}

// The thread's parameter is a handle to a pipe object instance.
hPipe = (HANDLE)lpvParam;

// Loop until done reading
while (1) {
      // Read client requests from the pipe. This simplistic code only allows messages
      // up to BUFSIZE characters in length.
      fSuccess = ReadFile(
            hPipe,   // handle to pipe
            pchRequest,  // buffer to receive data
            BUFSIZE,  // size of buffer
            &cbBytesRead, // number of bytes read
            NULL);

      if (!fSuccess || cbBytesRead == 0) {
            break;
      }

      // Process the incoming message.
      GetAnswerToRequest(pchRequest, pchReply, &cbReplyBytes);

      // Write the reply to the pipe.
      fSuccess = WriteFile(
            hPipe,   // handle to pipe
            pchReply,  // buffer to write from
            cbReplyBytes, // number of bytes to write
            &cbWritten,  // number of bytes written
            NULL);   // not overlapped I/O

      if (!fSuccess || cbReplyBytes != cbWritten) {
            break;
```

```
            }
      }

      // Flush the pipe to allow the client to read the pipe's contents
      // before disconnecting. Then disconnect the pipe, and close the
      // handle to this pipe instance.
      FlushFileBuffers(hPipe);
      DisconnectNamedPipe(hPipe);
      CloseHandle(hPipe);

      HeapFree(hHeap, 0, pchRequest);
      HeapFree(hHeap, 0, pchReply);
      return 1;
}

// This routine is a simple function to print the client request to the console
// and populate the reply buffer with a default data string. This is where you
// would put the actual client request processing code that runs in the context
// of an instance thread. Keep in mind the main thread will continue to wait for
// and receive other client connections while the instance thread is working.
VOID GetAnswerToRequest(char *pchRequest, char *pchReply, LPDWORD pchBytes) {
      printf("Client␣Request␣String:\"%s\"\n", pchRequest);

      // Check the outgoing message to make sure it's not too long for the buffer.
      if (FAILED(StringCchCopy(pchReply, BUFSIZE, "This␣is␣the␣answer."))) {
            *pchBytes = 0;
            pchReply[0] = 0;
            return;
      }
      *pchBytes = lstrlen(pchReply) + 1;
}
```

Listing D.2: NamedPipeClient.cpp

```
// Example code from: https://msdn.microsoft.com/en-us/library/windows/desktop/aa365592(v=vs.85).
    ↪ aspx

#include <Windows.h>
#include <stdio.h>
#include <conio.h>

#define BUFSIZE 512

int main(int argc, char *argv[]) {
      HANDLE hPipe;
      char* lpvMessage = "This␣is␣a␣test.";
      char chBuf[BUFSIZE];
      BOOL fSuccess = FALSE;
      DWORD cbRead, cbToWrite, cbWritten, dwMode;
      char* lpszPipename = "\\\\.\\pipe\\mynamedpipe";

      if (argc > 1)
```

```
      lpvMessage = argv[1];

// Try to open a named pipe; wait for it, if necessary.
while (1) {
      hPipe = CreateFile(
            lpszPipename, // pipe name
            GENERIC_READ | // read and write access
            GENERIC_WRITE,
            0,    // no sharing
            NULL,   // default security attributes
            OPEN_EXISTING, // opens existing pipe
            0,    // default attributes
            NULL);   // no template file

      // Break if the pipe handle is valid.
      if (hPipe != INVALID_HANDLE_VALUE)
            break;

      // Exit if an error other than ERROR_PIPE_BUSY occurs.
      if (GetLastError() != ERROR_PIPE_BUSY) {
            return -1;
      }

      // All pipe instances are busy, so wait for 20 seconds.
      if (!WaitNamedPipe(lpszPipename, 20000)) {
            return -1;
      }
}

// The pipe connected; change to message-read mode.
dwMode = PIPE_READMODE_MESSAGE;
fSuccess = SetNamedPipeHandleState(
      hPipe,  // pipe handle
      &dwMode, // new pipe mode
      NULL,  // don't set maximum bytes
      NULL);  // don't set maximum time

if (!fSuccess) {
      return -1;
}

// Send a message to the pipe server.
cbToWrite = (lstrlen(lpvMessage) + 1);

fSuccess = WriteFile(
      hPipe,   // pipe handle
      lpvMessage,  // message
      cbToWrite,  // message length
      &cbWritten,  // bytes written
      NULL);   // not overlapped

if (!fSuccess) {
```

```
            return -1;
      }

      do {
            // Read from the pipe.
            fSuccess = ReadFile(
                  hPipe,  // pipe handle
                  chBuf,  // buffer to receive reply
                  BUFSIZE, // size of buffer
                  &cbRead, // number of bytes read
                  NULL);

            if (!fSuccess && GetLastError() != ERROR_MORE_DATA)
                  break;

      } while (!fSuccess); // repeat loop if ERROR_MORE_DATA

      if (!fSuccess) {
            return -1;
      }

      getch();
      CloseHandle(hPipe);

      return 0;
}
```

D.2 Experiment 2

In the experiment 2, two clients run the same program in sequence to connect to the server with asynchronous Named pipe channel. The first piece of code listed below is the code for the server's program while the second piece is the test.bat is the script for running the experiment. The client program's code is identical to experiment 1.

Listing D.3: NamedPipeServerOverlapped.cpp

```
#include <Windows.h>
#include <stdio.h>
#include <strsafe.h>

#define CONNECTING_STATE 0
#define READING_STATE 1
#define WRITING_STATE 2
#define INSTANCES 4
#define PIPE_TIMEOUT 5000
#define BUFSIZE 4096

unsigned int ReplyCount = 0;
```

```
typedef struct {
      OVERLAPPED oOverlap;
      HANDLE hPipeInst;
      char chRequest[BUFSIZE];
      DWORD cbRead;
      char chReply[BUFSIZE];
      DWORD cbToWrite;
      DWORD dwState;
      BOOL fPendingIO;
} PIPEINST, *LPPIPEINST;

VOID DisconnectAndReconnect(DWORD);
BOOL ConnectToNewClient(HANDLE, LPOVERLAPPED);
VOID GetAnswerToRequest(LPPIPEINST);
PIPEINST Pipe[INSTANCES];
HANDLE hEvents[INSTANCES];

int main(VOID)
{
      DWORD i, dwWait, cbRet, dwErr;
      BOOL fSuccess;
      LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");
      // The initial loop creates several instances of a named pipe
      // along with an event object for each instance. An
      // overlapped ConnectNamedPipe operation is started for
      // each instance.
      for (i = 0; i < INSTANCES; i++)
      {
            // Create an event object for this instance.
            hEvents[i] = CreateEvent(
                  NULL, // default security attribute
                  TRUE, // manual-reset event
                  TRUE, // initial state = signaled
                  NULL); // unnamed event object

            if (hEvents[i] == NULL)
            {
                  return 0;
            }

            Pipe[i].oOverlap.hEvent = hEvents[i];
            Pipe[i].hPipeInst = CreateNamedPipe(
                  lpszPipename, // pipe name
                  PIPE_ACCESS_DUPLEX | // read/write access
                  FILE_FLAG_OVERLAPPED, // overlapped mode
                  PIPE_TYPE_MESSAGE | // message-type pipe
                  PIPE_READMODE_MESSAGE | // message-read mode
                  PIPE_WAIT, // blocking mode
                  INSTANCES, // number of instances
                  BUFSIZE*sizeof(TCHAR), // output buffer size
                  BUFSIZE*sizeof(TCHAR), // input buffer size
```

```
            PIPE_TIMEOUT, // client time-out
            NULL); // default security attributes

      if (Pipe[i].hPipeInst == INVALID_HANDLE_VALUE)
      {
            return 0;
      }

      // Call the subroutine to connect to the new client
      Pipe[i].fPendingIO = ConnectToNewClient(Pipe[i].hPipeInst, &Pipe[i].oOverlap);
      Pipe[i].dwState = Pipe[i].fPendingIO ? CONNECTING_STATE : READING_STATE;
}

while (1)
{
      // Wait for the event object to be signaled, indicating
      // completion of an overlapped read, write, or
      // connect operation.
      dwWait = WaitForMultipleObjects(
            INSTANCES, // number of event objects
            hEvents, // array of event objects
            FALSE, // does not wait for all
            INFINITE); // waits indefinitely

      // dwWait shows which pipe completed the operation.
      i = dwWait - WAIT_OBJECT_0; // determines which pipe
      if (i < 0 || i >(INSTANCES - 1))
      {
            printf("Index␣out␣of␣range.\n");
            return 0;
      }

      // Get the result if the operation was pending.
      if (Pipe[i].fPendingIO)
      {
            fSuccess = GetOverlappedResult(
                  Pipe[i].hPipeInst, // handle to pipe
                  &Pipe[i].oOverlap, // OVERLAPPED structure
                  &cbRet, // bytes transferred
                  FALSE); // do not wait

            switch (Pipe[i].dwState)
            {
                  // Pending connect operation
            case CONNECTING_STATE:
                  if (!fSuccess)
                  {
                        return 0;
                  }
                  Pipe[i].dwState = READING_STATE;
                  break;
                  // Pending read operation
```

```
      case READING_STATE:
            if (!fSuccess || cbRet == 0)
            {
                  DisconnectAndReconnect(i);
                  continue;
            }
            Pipe[i].cbRead = cbRet;
            Pipe[i].dwState = WRITING_STATE;
            break;
            // Pending write operation
      case WRITING_STATE:
            if (!fSuccess || cbRet != Pipe[i].cbToWrite)
            {
                  DisconnectAndReconnect(i);
                  continue;
            }
            Pipe[i].dwState = READING_STATE;
            break;
      default:
      {
            return 0;
      }
      }
}

// The pipe state determines which operation to do next.
switch (Pipe[i].dwState)
{
      // READING_STATE:
      // The pipe instance is connected to the client
      // and is ready to read a request from the client.
case READING_STATE:
      fSuccess = ReadFile(
            Pipe[i].hPipeInst,
            Pipe[i].chRequest,
            BUFSIZE*sizeof(TCHAR),
            &Pipe[i].cbRead,
            &Pipe[i].oOverlap);

      // The read operation completed successfully.
      if (fSuccess && Pipe[i].cbRead != 0)
      {
            Pipe[i].fPendingIO = FALSE;
            Pipe[i].dwState = WRITING_STATE;
            continue;
      }

      // The read operation is still pending.
      dwErr = GetLastError();
      if (!fSuccess && (dwErr == ERROR_IO_PENDING))
      {
            Pipe[i].fPendingIO = TRUE;
```

```
                        continue;
                    }

                    // An error occurred; disconnect from the client.
                    DisconnectAndReconnect(i);
                    break;

                    // WRITING_STATE:
                    // The request was successfully read from the client.
                    // Get the reply data and write it to the client.
                case WRITING_STATE:
                    GetAnswerToRequest(&Pipe[i]);

                    fSuccess = WriteFile(
                        Pipe[i].hPipeInst,
                        Pipe[i].chReply,
                        Pipe[i].cbToWrite,
                        &cbRet,
                        &Pipe[i].oOverlap);

                    // The write operation completed successfully.
                    if (fSuccess && cbRet == Pipe[i].cbToWrite)
                    {
                        Pipe[i].fPendingIO = FALSE;
                        Pipe[i].dwState = READING_STATE;
                        continue;
                    }

                    // The write operation is still pending.
                    dwErr = GetLastError();
                    if (!fSuccess && (dwErr == ERROR_IO_PENDING))
                    {
                        Pipe[i].fPendingIO = TRUE;
                        continue;
                    }

                    // An error occurred; disconnect from the client.
                    DisconnectAndReconnect(i);
                    break;

                default:
                {
                    return 0;
                }
                }
        }

        return 0;
}

// DisconnectAndReconnect(DWORD)
// This function is called when an error occurs or when the client
```

```
// closes its handle to the pipe. Disconnect from this client, then
// call ConnectNamedPipe to wait for another client to connect.
VOID DisconnectAndReconnect(DWORD i)
{
      // Disconnect the pipe instance.
   DisconnectNamedPipe(Pipe[i].hPipeInst)
      // Call a subroutine to connect to the new client.
      Pipe[i].fPendingIO = ConnectToNewClient(Pipe[i].hPipeInst, &Pipe[i].oOverlap);
      Pipe[i].dwState = Pipe[i].fPendingIO ? CONNECTING_STATE : READING_STATE;
}

// ConnectToNewClient(HANDLE, LPOVERLAPPED)
// This function is called to start an overlapped connect operation.
// It returns TRUE if an operation is pending or FALSE if the
// connection has been completed.
BOOL ConnectToNewClient(HANDLE hPipe, LPOVERLAPPED lpo)
{
      BOOL fConnected, fPendingIO = FALSE;

      // Start an overlapped connection for this pipe instance.
      fConnected = ConnectNamedPipe(hPipe, lpo);
      // Overlapped ConnectNamedPipe should return zero.
      if (fConnected) {
            return 0;
      }

      // Sleep random time for overlap
      Sleep(1000 * (1 + rand() % 4));

      switch (GetLastError()) {
      // The overlapped connection is in progress.
      case ERROR_IO_PENDING:
            fPendingIO = TRUE;
            break;
      // Client is already connected, so signal an event
      case ERROR_PIPE_CONNECTED:
            if (SetEvent(lpo->hEvent))
                  break;
      // If an error occurs during the connect operation...
      default:
      {
            return 0;
      }
      }
      return fPendingIO;
}

void GetAnswerToRequest(LPPIPEINST pipe)
{
      unsigned int currentCount = ReplyCount;
      ReplyCount++;
      StringCchCopy(pipe->chReply, BUFSIZE, "Answer␣from␣server");
```

```
      pipe->cbToWrite = lstrlen(pipe->chReply) + 1;
}
```

Listing D.4: test.bat

```
@echo off
start "Server" NamedPipeServerOverlapped.exe

start "Client 1" NamedPipeClient.exe "Message 1"
start "Client 2" NamedPipeClient.exe "Message 2"
```