Parsing and Analyzing POSIX API behavior on different platforms

Savvas Savvides

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science
Department of Computer Science

New York University
May 2013

Prof. Justin Cappos

Prof. Jinyang Li

Acknowledgements

I wish to express my gratitude to my research advisor, Professor Justin Cappos who was abundantly helpful and offered invaluable assistance and guidance. I would like to thank Professor Phyllis Frankl for her support in this project and Professor Jinyang Li, my second reader. Thanks also to my fellow student Eleni Gessiou for some interesting and very helpful conversations regarding this project, without which this thesis would not have been the same.

Abstract

Because of the increased variety of operating systems and architectures, developing applications that are supported by multiple platforms has become a cumbersome task. To mitigate this problem, many portable APIs were created which are responsible for hiding the details of the underlying layers of the system and they provide a universal interface on top of which applications can be built. Many times it is necessary to examine the interactions between an application and an API, either to check that the behavior of these interactions is the expected one or to confirm that this behavior is the same across platforms.

In this thesis, the *POSIX Omni Tracer* tool is described. *POSIX Omni Tracer* provides an easy way to analyze the behavior of the *POSIX API* under various platforms. The behavior of the *POSIX API* can be captured in traces during an application's execution using various tracing utilities. These traces are then parsed into a *uniform representation.* Since the captured behavior from different platforms share the same format, they can be easily analyzed or compared between them.

Contents

1 Introduction

In the past few years there has been a steep rise in the mobile industry, increasing the diversity of platforms on which software is built. With several new operating systems recently introduced (Android OS[17], Firefox OS[16, 1], Ubuntu Mobile OS[19], Chrome OS[18]) and the traditional operating systems on laptops and desktops remaining popular, there are nowadays a significant number of operating systems in wide use.

Application developers strive to deploy their applications on as many of these platforms as possible, hence reaching more customers. However, the increased diversity of operating systems and architectures makes this task hard to achieve. Over the years, several APIs were introduced, aiming to provide a portable interface and make the task of porting applications in multiple platforms easier and less time consuming. The job of these APIs is to abstract the irregularities of the underlying layers and provide a universal interface for applications. To do this, an API must mask the behavior of the operating system, the file system and the lower level libraries of the platform. The Portable Operating System interface (*POSIX*) is one such API, aiming to promote portability amongst operating systems that support it. It does so by exposing an interface of many system calls, through which an application can interact with the operating system.

The aim of the work described in this thesis was to come up with a way to describe the behavior of the *POSIX API* during an application's execution, so that it could be examined and analyzed through the use of plug-ins or external applications. Furthermore, a way had to be found that would allow this on different platforms running different Operating Systems. Once this was achieved, a mechanism had to be provided to reform different instances of the *POSIX API* behavior into a uniform representation, which would have the same format regardless of the underlying platform that behavior was captured on. Using this uniform representation, instances of *POSIX API* behavior from different platforms could not only be analyzed individually, but also compared between them, which could potentially help in porting an application into different platforms.

In this thesis, the development of the *POSIX Omni Tracer* (*POT*) tool is described. *POT* is able to capture and reform the system calls involved in the interactions between an application and the *POSIX API*, on different platforms. Various tracing utilities are used to record these interactions, which are then combined with some additional environment state information (section 6) to accurately capture the behavior of the *POSIX API*. *POT* parses this behavior and generates a uniform representation which is then used to analyze the behavior of the *POSIX API*.

The structure of this thesis is devided as follows. Firstly, an overview of the related work is given in section 2. Then the general design of *POT* is explained in section 3. Section 4 describes the method used to trace the *POSIX API* behavior, followed by section 5 which explains the mechanism used to parse the behavior captured from different systems into a uniform representation. In section 6 the way the environment state is stored is discussed. Then, section 7 shows ways in which the captured behavior can be analyzed and used to extract useful information. The evaluation of this work is given in section 8 before concluding in section 9.

2 Related Work

The related work of this project is given in two parts. Firstly, the existing mechanisms for tracing the *POSIX API* behavior are discussed. Then, a set of tools that are able to parse this behavior are described and compared to the functionality of *POT*.

2.1 Tracing POSIX API behavior

Tracing the system calls describing the behavior of a POSIX API execution can be trivially achieved under Linux and Solaris platforms, through the use of the *strace* and the *truss* utilities. But tracing system calls under BSD and OSX platforms is not as straightforward. Details on how this is done are given in section 4.3 and they involve using a *D script* with the *dtrace* framework. One such *D script* is *dtruss* [5]. *dtruss* tries to imitate the behavior of the *truss* utility (hence the name) by producing results that resemble the *truss* results. *dtruss* is limited in that it does not capture all the information needed to fully describe a system call (many argument values are missing). The *D script* developed as part of this work deals with these limitations by capturing all the information necessary to fully describe a system call.

2.2 Parsing and Analyzing System Calls

Stana or the STrace ANAlyser [3] is a tool that can parse output trace files generated from the *strace* utility on Linux platforms. It uses native plug-ins on top of a parser to provide statistics about the trace. It currently supports plug-ins for summarizing the file I/O activity and tracking the children processes of the main process.

Pystrace [2] is another parser written in python that can parse the system calls from the output of the *strace* utility. The parsed system calls are then stored in a *cvs* file. Additionally, *pytrace* can optionally track and provide time statistics (start/stop/elapsed time) for each process that appears in the trace.

The *strace_analyzer* [11] is another tool that explores information taken from *strace* output files. This tool handles only very few I/O system calls and focuses on providing statistics through which one can discover patterns related to file accessing. For example, it provides information on how many files are read or written and the amount of data read from or written to a file.

IOapps [7] is a very useful tool suite that provides the *ioreplay* and the *ioprofiler* tools. As the names might suggest, these tools are used to perform certain I/O related operations. Both of these tools are build on top of the same code base, which records the behavior of a set of approximately 20 I/O related system calls. Like all the previous tools discussed so far, *IOapps* collects these information by parsing *strace* traces observed during an application's execution. Subsequently, the *ioreplay* tool uses the recorded system calls to simulate (replay) the behavior of the traced application. This is useful for example when there is a need to carry out I/O benchmarks for an application on a system and for some reason the application itself cannot be executed directly on that system. The *ioprofiler* tool is used to provide statistics similar to the ones provided by the *strace_analyzer*, but it does so through the use of graphical user interface and uses plots to provide the results.

*(*POT) differs significantly from other tools. Firstly, *(*POT) has a widely usable parser which can be used to parse traces coming from different utilities on different operating systems. The input format it takes is not limited to *strace*. *POT*can also support input from *strace*, *truss* and *dtrace*. This allows *(*POT) to analyze the POSIX API behavior traced from a wide range of platforms, as long as they support one of these utilities.

Furthermore, *(*POT) is not limited in dealing with only I/O related system calls. It is a generic parser that can handle system calls of different classifications. In fact, one of the main aims when developing *(*POT), was to provide support for the most commonly used system calls, regardless of their classification or usage. As a result, this tool can handle system calls related to file access and file descriptor operations but it can also handle system calls used for network access, process control and system control.

Most notably, *(*POT) does not merely parse a system call into its different parts (name, arguments, return values). Instead it smartly extracts information from the system call into a meaningful uniform representation, with each argument and return value associated with a special class indicating its type. Because of the generic nature of *(*POT), and the resulting uniform representation of the parsed traces, *(*POT) is not only useful in providing statistical I/O information but can also act as the base for other applications to provide system support, network support and security support.

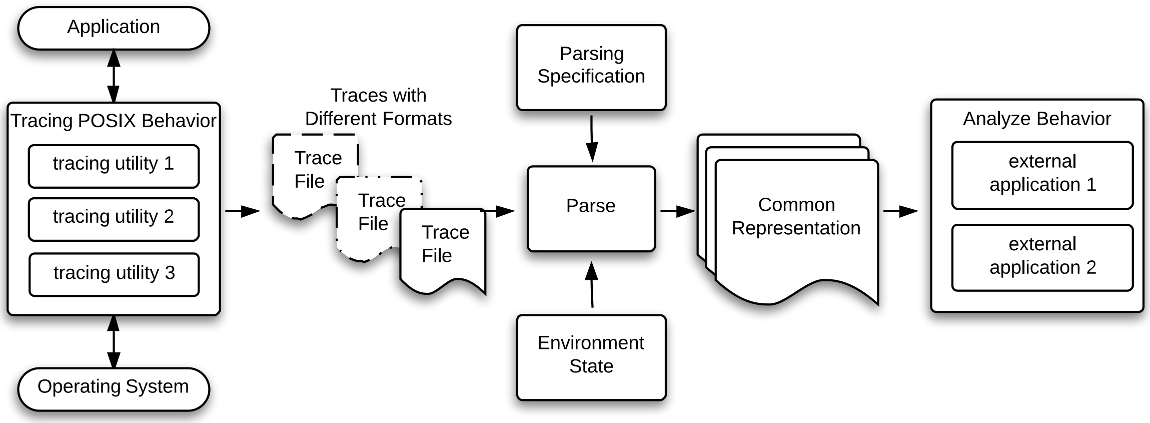

Figure 1: The components that make up the *POT* tool.

3 POSIX Omni Tracer

The *POSIX Omni Tracer* is a universal tracer that can capture the behavior of the *POSIX API* on many platforms. It currently supports any Linux, Solaris, BSD and OSX platforms and potentially it can be expanded to support even more platforms. The *POSIX API* behavior is captured in trace files using various tracing utilities. Because these trace files come from different utilities, they have a different format. This means that an application that wants to make use of the behavior of the *POSIX API*, would have to provide a different mechanism to capture this behavior on each of the platforms it needs to support. *POT* solves this problem by parsing these trace files and generating a uniform representation. This representation extracts all the useful information from the trace files but masks format and representation differences. The uniform representation can then be used as a generic framework on top of which other applications can be built.

The current implementation of *POT* is able to intercept and examine the 55 most commonly used system calls. These system calls were chosen regardless of their classification, with the aim of describing the *POSIX API* behavior as comprehensively as possible. Support for more system calls is constantly added making *POT* a more complete system. (*POT* initially supported around 30 system calls and it was gradually extended to support more system calls). Additionally, every time *POT* is used, it makes a note of how many times each system call (out of all system calls) is met, so that it can identify which other system calls are frequently used and designate which are the next system calls to add support for.

3.1 Design

Figure 1 outlines the general design of *POT*. Whenever an application is executed it will interact with the operating system through the *POSIX API system call interface*. The set of all system calls called during the execution of this application can be traced using a trace utility (see section 4). Different operating systems support different tracing utilities hence depending on the operating system the application was executed on, different tracing utilities will be used. This will result in trace files of different formats. The next step is to take these trace files and parse them into a uniform and common representation. This is achieved through a parsing mechanism, that takes as input a trace file of any format and uses a parsing specification to generate a uniform representation. In some cases the parser will use information from the environment state in which the application was executed, so that it can accurately capture the *POSIX API* behavior. Once the common representation is generated, external applications can be developed that make use of this representation to analyze the behavior of *POSIX API*.

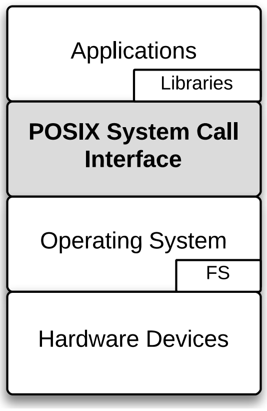

Figure 2: The POSIX System Call Interface

4 Tracing *POSIX API* behavior

This section is divided in three parts. Firstly, a brief description is given about the *POSIX API* and the POSIX system call interface. Then an explanation is given about traces and how they can be used to capture the behavior of the *POSIX API*. Finally an extensive analysis is given on how traces can be gathered on each platform.

4.1 The POSIX API

The Portable Operating System interface (POSIX) is a portable application program interface aiming to promote portability amongst operating systems. More formally, a POSIX-compliant application should be able to run on any POSIX- compliant operating system without the need of changing its source. Several operating systems are considered POSIX-compliant, with the most prominent of them being Linux, Solaris, BSD and OSX. POSIX-compliant applications interact with the operating system through the POSIX system call interface which is a collection of system calls, as shown in figure 2. The behavior of these interactions should be the same across all POSIX-compliant platforms.

4.2 Traces

The *POSIX API* behavior can be specified in the form of traces. Every time the API is executed a trace can be recorded describing the behavior of the API during that execution. A trace is a collection of atomic units of behavior, called actions, where each action represents the execution of a single system call. Actions are defined in terms of three components. These components are the action name, which indicates which method was executed, the action parameters (these are not necessarily the same as the system call parameters, as explained in section 5.5), which directly affect the action's behavior and the action result which indicates whether the action was successful or not. The action result can also include the return values of the system call and information about errors that occurred.

action = (action_name, action_arguments, action_result)

Sometimes a system call can have side-effects that are not reflected in the action. For example, when an *open* system call is executed, it is possible that a new file will be created. Any side-effects caused by an API call which are not recorded in the action directly, will become apparent at a later point, by observing the resulting behavior of subsequent actions. Alternatively, some side-effects that are not realized throughout the trace will be captured later, when the environment information is recorded (see section 6).

4.3 Trace Capturing

One of the main aims of POT was to capture the behavior of the *POSIX API* under the most commonly used POSIX-compliant platforms. This behavior should only contain the interactions between the *POSIX API* and the application executed. The capturing mechanism itself, should not interfere with the API's behavior nor should interactions with other applications and the *POSIX API* be recorded in the trace. Therefore, a way had to be found to generate a trace action every time a system call was executed, and also do this in all the main POSIX-compliant platforms like Linux, Solaris, BSD and OSX. Furthermore, each action had to include the values of all the arguments the system call used and also its return status and its return values. Fortunately, many incarnations of Unix platforms provide tools that allow the tracing of kernel system calls.

4.3.1 Capturing traces in Linux platforms.

Capturing traces in Linux platforms can be achieved by using the *strace* (system trace) utility[12]. *strace* takes as arguments a command. It then runs the given command and makes use of a specialized kernel system call, called *ptrace* [10] which allows it to observe and intercept all the system calls called by that command. Then *strace* prints the name of each intercepted system call, followed by its arguments and its return values to the standard error or to a file provided as an option to *strace*. More importantly, *strace* includes a verbose option which tells it to capture all the passed arguments and return values of a system call unabbreviated. *strace* proved to be more than sufficient in providing the information needed to form traces under Linux.

Not only does *strace* capture all the information that POT requires, but it does so in a very concise and readable output format. Figure 3 shows an example of strace output. Each line represents a recorded system call. Firstly the id of the process in which the system call was executed is given. Then the name of the system call is given followed by the arguments passed to the system call enclosed within brackets. Finally, the result of the system call is given right after the "=" sign.

In this example, an *open* system call is firstly recorded. This system call takes as arguments the string "file.txt" which is the path to the file to open, and the *RD_ONLY* flag telling it that the file needs to be opened only for reading. The system call succeeds and returns a file descriptor (namely 3), pointing to the file just opened. In line 2, a *read* system call is issued on this file descriptor, and the text "data" is read from "file.txt". Note here

```
1  28433 open("file.txt", O_RDONLY)  = 3
2  28433 read(3, "data", 32768)      = 4
3  28433 write(1, "data", 4)         = 4
4
5  28433 open("file2.txt", O_RDONLY) = −1 ENOENT
```

Figure 3: Example of *strace* output. Lines 1 to 3 show a set of successful system calls and line 5 shows a system call that failed with an *ENOENT* error.

that even though the text read from the file is the result of this system call, in the *strace* output it is given as one of the input arguments of the system call. The way POT handles this irregularity is discussed in section 5.5. In line 3, the text read from the file is written in the standard output (file descriptor 1). The return part for both *read* and *write* system calls indicates the number of characters read or written respectively. Overall, the system calls displayed in lines 1 to 3 represent the strace output when tracing the command "cat file.txt".

Line 5 shows an attempt to open another file called "file2.txt". In this case the *open* system call fails, and instead of a valid file descriptor, the number -1 is returned. This number is followed by the error label *ENOENT* which stands for "Error no entry" and it indicates that the file attempting to open does not exist. There are several other error labels like this one with the most common ones being *EACCES* which indicates that access to the requested file is not permitted and *EEXIST* which indicates that the file trying to create already exists.

All system calls share the same convention on what the format of their return part should be. If the system call is executed correctly, a non-negative number is returned. But if an error occurs during the execution of the system call, then the system call will return -1 followed by an error label. To be precise, not all system calls return -1 to indicate an error per se, but at the end this convention is followed by all system calls nevertheless. This is because of the way system calls are invoked. Generally, system calls are not called directly but rather invoked through a *system call wrapper*. When a system call has an error, it will return a negative error number (not necessarily -1) back to the system call wrapper. When this happens, the wrapper negates the returned error number to make it positive and copies this number to the *errno* special variable. Then the system call wrapper returns -1. Subsequently, when *strace* or another tracing utility intercepts such system call, it will see -1 as the return value and it will check the value of the *errno* variable and translate it to the appropriate error label. Therefore, the convention of returning -1 in case of an error is seemingly followed by all system calls.

4.3.2 Capturing traces in Solaris platforms.

Solaris platforms do not have the *strace* utility but instead they provide a utility called *truss* [13]. *truss* stands for "trace user system calls and signals" and it can execute a specified command and produce a trace containing all the system calls performed by that command. It prints a line for each system call traced, including its arguments and return

```
1  ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
2                  struct sockaddr *src_addr, socklen_t *addrlen);
3
4  2535:   recvfrom(3, 0xD1D3EDA0, 512, 0, 0xD1D3EFB0, 0xD1D3EFA8) = 20
5  2535:      T r a n s m i t t e d   M e s s a g e\0
6  2535:     AF_INET  from = 127.0.0.1  port = 33035
```

Figure 4: Example of *truss* output. The definition of *recvfrom* is given in lines 1 and 2 followed by the *truss* output for the intercepted *recvfrom* system call.

values. Additionally, *truss* makes use of a set of system headers to display the arguments of system calls symbolically where possible. For example numeric values representing flags are replaced with the name of that flag. Similarly to *strace*, *truss* provides options allowing it to record the input and output strings in read and write system calls respectively. It also has a verbose option which if selected, it tells *truss* to dereference all the structures involved in the system calls, and provide their values.

When using the *truss* utility to gather traces on Solaris platforms, a shortcoming of *truss* was observed. Even though no documentation was found to confirm this issue, it seems that pointers to integer values are not dereferenced and instead, the address of the value is returned. Figure 4 shows an example of *truss* output which demonstrates this shortcoming. In lines 1 and 2, the definition of the *recvfrom* system call is given, and below it, lines 4 to 6 show the *truss* output format for tracing this system call. The socket file descriptor is given in line 4 and it is equal to 3. The value of the buffer containing the received message is given in line 5. The length of the buffer is given as 512 and the socket address structure is given in line 6. The last argument though, which is the length of the socket address structure is a pointer to an integer value (*socklen_t ** to be precise), and its value is not dereferenced. Instead, *truss* returns its address *0xD1D3EFA8*. To deal with this issue, *POT* makes appropriate arrangements in the parsing phase, described in section 5.3. In essence, the action with the missing value is marked so that a warning can be issued when an application uses this action.

4.3.3 Capturing traces in BSD platforms.

Unfortunately, BSD platforms do not provide the *strace* utility and the truss utility they include is a limited version of the one used in Solaris. *truss* on BSD fails to provide a verbose option and as a consequence, not all structure values are displayed. For example, only four values of the stat structure (used in stat system call) are displayed.

Since *strace* and *truss* could not be used, another way of gathering traces under BSD systems had to be found. Firstly the *ktrace* [8] utility was considered as a replacement. *ktrace* is a much more comprehensive utility compared to the *truss* implementation on BSD and it provides most of what *POT* requires. For almost all the system calls *POT* supports, *ktrace* dereferences and provides the required content. Unfortunately, *ktrace* fails to provide the values of some structures in certain system calls. As an example, figure 5 shows the *ktrace* output for the *statfs* system call. *statfs* is usually used to get information about a

```
1   22205 syscalls CALL  stat(0x804df40,0xbfbfeb68)
2   22205 syscalls NAMI  "syscalls.txt"
3   22205 syscalls STRU  struct stat {dev=76, ino=151953, mode=-rw-r--r-- , nli
   nk=1, uid=0, gid=0, rdev=0, atime=1367637329, stime=1367637590, ctime=136763
   7590, birthtime=1367637329, size=0, blksize=32768, blocks=0, flags=0x0 }
4   22205 syscalls RET   stat 0
5
6   22213 syscalls CALL  statfs(0x804df40,0xbfbfe9f0)
7   22213 syscalls NAMI  "syscalls.txt"
8   22213 syscalls RET   statfs 0
```

Figure 5: Example *ktrace* output. Lines 1 to 4 show the ktrace output for the stat system call with the values of its structure displayed. Lines 6 to 8 show the ktrace output for the statfs system call with its structure not being dereferenced.

file system. It takes as arguments a statfs structure and if the system call succeeds, the structure values are filled and returned. In the *ktrace* output shown in this figure, even if the system call is successful, the statfs structure is not dereferenced and instead only its address is recorded in the output file. Even though it provides most of the content needed by *POT*, the few omissions it has, render *ktrace* unsuitable for the purposes of *POT*.

As a final choice *dtrace* [4] was considered. *dtrace* (Dynamic Tracing) is not a tracing utility like the other ones described above, but rather a framework used for real time system analysis. It provides a wide variety of features which can be explored through the use of scripts, written in the *dtrace D language*, called D scripts. The D language is a subset of the C language but it also includes functions and variables specific to tracing. Scripts in this language make use of a set of probes provided by *dtrace*. Each probe represents an action, and whenever the conditions of that action are met, the probe executes. Fortunately, *dtrace* provides a probe for every system call that exists in the *BSD* platforms, and hence it was adopted as the tracing utility used for gathering traces on BSD systems for this project.

A *D script* containing two probes for each of the 55 system calls supported by *POT* was written. An "entry" probe fires once a system call is called and a "return" probe fires when that system call returns (or when an error occurs). As shown in figure 6, within the "entry" probe, the parameters of the system call are intercepted and recorded. The definition of the probe given in line 7, indicates that this probe should only fire when the system call with name *connect* is called. Following this definition, a predicate is given. Predicates work as conditions through which additional restrictions can be imposed on when a probe is fired. In this case, line 8 specifies that the entry probe of *connect* should only fire when the pid of the process that executed this system call is not the pid of the *dtrace D script* itself. This way the traced behavior will not contain actions from any other process except the one of the application being traced. In line 11, a variable named "start" is initialized. This variable will be used as a predicate for the return probe. In line 14, the socket file descriptor of the *connect* system call is recorded.Similarly, in line 17 the pointer to the *sockaddr* structured is recorded and in line 20 the length of this structure is stored.

Subsequently, within the "return" probe shown in figure 7 the return values and error messages are captured and the entire action printed. Line 2 shows the predicate for this

```
 1  int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
 2
 3  /*
 4   * Entry probe for the connect system call. This probe fires when the
 5   * connect system call is called
 6   */
 7  syscall::connect:entry
 8  /pid != $pid/ // predicate: do not trace the dtrace process ($pid)
 9  {
10    // mark the system call as started
11    self->start = timestamp;
12
13    // save the socket file descriptor
14    self->sockfd = arg0;
15
16    // save the pointer to the sockaddr structure
17    self->sockaddr_pointer = arg1;
18
19    // save the size of the sockaddr structure
20    self->sockaddr_len = arg2;
21  }
```

Figure 6: Dtrace entry probe for the connect system call. For clarity line 1 gives the definition of the *connect* system call. (Line 1 is not part of the D script.)

probe. Apart from checking the pid, this predicate checks if the start variable was set in the entry probe. This ensures that the return probe will only fire if the entry probe was fired first. Lines 5 to 18 show how the sockaddr structure is dereferenced. In lines 20 to 28 the process id and the system call name and arguments are printed. Finally, in lines 30 to 32, the *errno* special variable is checked, and if an error occurred in the execution of the system call, an error label is printed.

A final complication that had to be dealt with before gathering traces in BSD platforms, was the fact that the *dtrace* framework is not supported by the standard (*GENERIC*) BSD kernel. To overcome this issue, the kernel had to be customized in order to support dtrace before the latter could be used to gather traces. Capturing the behavior of the *POSIX API* under BSD platforms took more work compared to the other operating systems. But because of the impressive array of features *dtrace* provides, it allowed for an output that contained all the information *POT* needed. In addition, due to the flexibility of dtrace, the output result was formatted appropriately so that it could be handled easily when it was next needed.

4.3.4 Capturing traces in OSX platforms.

The last operating system *POT* had to provide support for was OSX. OSX (Darwin) is a BSD-derived operating system and it fully supports the *dtrace* framework (in fact, unlike BSD not even kernel customization is needed to run *dtrace*). As a result, the *D Script* used to gather traces on BSD could also be used to gather traces on OSX platforms. In reality, subtle differences between the two *dtrace* versions on these operating systems had to be addressed. For example *dtrace* on OSX had different names for some of the probes used in the BSD *D script*. Furthermore, in order for *dtrace* on OSX to be able to dereference structures and display their values, the structure definitions had to be explicitly defined (not

```
syscall::connect:return
/self->start && pid != $pid/ // only fire if entry probe has fired
{
  // get the sockaddr structure (dereference)
  this->sockaddr = (struct sockaddr*) copyin(self->sockaddr_pointer,
                                          sizeof(struct sockaddr));
  // get the socket family
  self->family = this->sockaddr->sin_family;
  // get the socket port
  self->port = ntohs(this->sockaddr->sin_port);

  // get the socket ip address
  this->a = (uint8_t *) &this->sockaddr->sin_addr;
  this->addr1 = strjoin(lltostr(this->a[0] + 0ULL),
                strjoin(".", strjoin(lltostr(this->a[1] + 0ULL), ".")));
  this->addr2 = strjoin(lltostr(this->a[2] + 0ULL),
                strjoin(".", lltostr(this->a[3] + 0ULL)));
  self->ip = strjoin(this->addr1, this->addr2);

  // print the process id
  printf("%d ", pid);
  // print the name of the probe function. This should be "connect"
  printf("%s", probefunc);
  // print the system call parameters
  printf("(%d, {sa_family=%d, sin_port=htons(%d), \
                sin_addr=inet_addr(\"%s\")}, %d)\t\t ",
         self->sockfd, self->family, self->port,
         self->ip, self->sockaddr_len);

  // translate error number to error name and print result
  this->errstr = err[errno] != NULL ? err[errno] : "";
  printf("= %d %s\n", (int) arg0, this->errstr);
}
```

Figure 7: Dtrace return probe for the connect system call.

imported) in the D script, as shown in figure 8.s

A small deficiency of *dtrace* as opposed to the *strace* and *truss* utilities, is that the former can only run under root privileges. This is true for both *dtrace* on BSD and *dtrace* on OSX. Fortunately, this did not cause any complications since all the applications whose behavior was traced could be executed as root.

```
 1  struct stat {
 2     dev_t    st_dev;
 3     ino_t    st_ino;
 4     mode_t   st_mode;
 5     nlink_t  st_nlink;
 6     uid_t    st_uid;
 7     gid_t    st_gid;
 8     dev_t    st_rdev;
 9     struct timespec st_atimespec;
10     struct timespec st_mtimespec;
11     struct timespec st_ctimespec;
12     off_t    st_size;
13     quad_t   st_blocks;
14     u_long   st_blksize;
15     u_long   st_flags;
16     u_long   st_gen;
17  };
18
19  22884 stat("syscalls.txt", {st_dev=76, st_ino=151831, st_mode=22188,
20             st_nlink=1, st_uid=0, st_gid=0, st_blksize=32768, st_blocks=0,
21             st_size=0, st_atime=1081885184, st_mtime=1081885184,
22             st_ctime=1081885184}) = 0
```

Figure 8: dtrace output. Lines 1 to 17 show the stat structure definition that needs to be explicitly copied in the D script. Lines 19 to 22 show the dtrace output for the intercepted stat system call.

```
1  # definition of the connect system call.
2  int connect(int sockfd, const struct sockaddr *addr,
3              socklen_t addrlen);
4
5  # Specification for the connect system call.
6  "connect": {
7    'arguments': (Int(), Flags(), Int(), Str(), Int()),
8    'return': (Int(), NoneOrStr())
9  }
```

Figure 9: Action specification for the *connect* system call.

5 Parsing POSIX API Behavior

Section 4 describes how traces are gathered from different operating systems. This is achieved through the use of various utilities that are able to interpose the interactions between an application and the operating system kernel, and intercept the system calls involved in that application's execution. Because different utilities are used in each operating system, the format of the traces captured in different platforms varies significantly. This creates a problem when trying to reason about the behavior of these traces. To mitigate this problem all the traces have to be parsed into a common format before they can be used.

5.1 Uniform Representation

In order to be able to examine the behavior of traces gathered in different platforms in a similar fashion, it was necessary to come up with a uniform representation to describe the intercepted actions. This representation would have to hide the format differences in traces gathered from different operating systems using different utilities. Hence, after each trace was captured, it would be parsed into a uniform representation. This representation includes all the required information contained in the trace parsed, with the format differences abstracted away. Therefore, after parsing traces into their uniform representation, it is impossible to tell which platform those traces were gathered from. As a result they can be easily used by other applications without the need of dealing with each trace format separately.

To achieve a uniform representation for all captured traces, an unambiguous format had to be specified for all the parts that make up a trace, or in other words, all the possible actions of a trace. For this reason, a precise *parsing specification* was defined. This parsing specification is made up by a set of *action specifications* with each system call having its own action specification. Action specifications specify the expected arguments and the expected return values for the system call they represent and they impose certain requirements on what their type and value should be. But apart from defining the way that the system calls should be parsed, action specifications are used to associate each argument and each return value of a system call with a special class that gives meaning to these values (see Section 5.3).

An example of such specification is shown in figure 9. In this example, lines 2 and 3 give the definition of the *connect* system call. *connect* is used to associate an address to a given socket. It takes as arguments a file descriptor of a socket (this can be taken using the *socket* system call), a reference to a *sockaddr* structure containing all the information needed to describe an address and an integer value (*socklen_t*) specifying the size of that structure. Then, lines 6 to 9 give the action specification of *connect*. This specification defines what types of arguments and what types of return values the trace must contain, whenever a connect system call is parsed. Regardless of which tool was used or which operating system a trace was gathered from, a *connect* action should have 5 arguments. Note that the sockaddr structure shown in the definition of connect is made up of three components. These components are the address family, the port and the IP. In trace files these components are given separately. This is why the definition of connect has three arguments and the specification of connect has five arguments. As shown in the specification, the first argument should be an integer value (*Int()*) which will hold the socket file descriptor. The second argument should be of type *Flags()* which will hold the socket address family e.g *AF_INET*. The third argument should be an integer to hold the port number of the sockaddr structure and the forth should be a string *Str()* to hold the IP. Finally, another integer argument is needed to hold the total size of the sockaddr structure.

The return values are also defined in the action specification as an *Int()* object followed by a *NoneOrStr()* object. The *Int(), NoneOrStr()* combination is used for the return part of all system calls. The return part of a system call is always a non-negative number when the system call succeeds or a -1 followed by the error label when it fails. The *Int(), NoneOrStr()* class combination tells the parser that it should expect an integer value followed by either a *None* value in case the system call succeeded or a *Str()* value to hold the error label in case the system call failed.

The types of all arguments and return values in the specification are defined as objects of a special class. In this example these classes are *Int()*, *Str()*, *Flags()* and *NoneOrStr()* but there are several others. Here *Int()* and *Str()* are not the same as the built in types *int* and *string*. The reason why these objects are used instead of the conventional built-in types is explained in section 5.3.

5.2 Hiding format differences.

Using the action specifications, a parser translates *strace*, *truss* and *dtrace* traces into the uniform representation. To do this, the parser has to deal with many format peculiarities of each trace type. Firstly, the parser has to rename some system calls so that actions representing the same system call have a consistent name. For example, Linux has a system call called *statfs*. In Solaris platforms, this system call uses the name *statvfs* (and uses the statvfs structure instead of the statfs structure). This type of irregularity happens for a few other system calls too (fstatfs-fstatvfs, stat-xstat, fstat-fxstat, getdents-getdirentries, etc). Since these system calls are conceptually the same, the parser makes sure they have the same name when parsed.

Another format difference the parser has to deal with is the fact that some *truss* traces include unnecessary values in the arguments of some system calls. An example where this format difference appears is the *stat* system call. *stat* is used to get statistics about a specific

```
1  4328  bind(3, {sa_family=AF_INET, sin_port=htons(25588),
                  sin_addr=inet_addr("127.0.0.1")}, 16) = 0
2
3  2535: bind(3, 0xD1D3EFC0, 16, SOV_SOCKBSD)    = 0
4  2535:     AF_INET  name = 127.0.0.1  port = 25588
5
6  ('bind_syscall', (3, ['AF_INET'], 25588, '127.0.0.1', 16), (0, None))
```

Figure 10: *strace* (line 1) and *truss* (lines 3 and 4) output comparison for the bind system call and the resulting uniform representation

file. It takes as arguments a path to a file and a reference to a stat structure which will hold the information to be returned. When this system call is traced using *truss*, an unnecessary third argument is included. This argument is removed when the system call is parsed.

Figure 10 shows a comparison between the *strace* and the *truss* output for the *bind* system call. The format of the two utilities is significantly different although the information they carry is of course the same. In *strace*, output for a system call is given in a single line, with structure values enclosed within angle brackets. In the *truss* output on the other hand, structure values are given in a new line. Once these two formats are parsed, the resulting uniform representation is the same for both cases. This example demonstrates another format difference, which is the fact that the "SOV_SOCKBSD" flag, which only appears in the *truss* output, is removed and excluded from the uniform representation.

Format differences on the output of traced actions appear mainly between *strace* and *truss* output. Because *dtrace* is flexible, its output format can be structured in the way needed within the return probe of the *D script* used to trace the system calls. Therefore, the probes used to capture the *POSIX API* behavior on BSD and OSX platforms, were written so that they produce an output with the same format as the one used in strace.

5.3 Masking content discrepancies.

Format differences are not the only peculiarities of trace files that the parser needs to deal with. The parser has to make sure that the content of actions of the same kind (actions that represent the same system call) will contain the same set of arguments and return values. To achieve this, a set of special classes were defined and used to make up the action specifications as described in section 5.1. As previously mentioned, each action is defined in terms of the action name, the action parameters and the action return values. Each individual argument and return value is defined in terms of one of these special classes. These classes would then enforce the expected type and format of each argument and return value of the system call.

Not only are these classes able to detect unexpected values, they are also able to detect when a argument or return value contains the right content but in a different format. In this case, the value will be translated to match the expected format. A case that such a translation is required is when the system call contains flags. For example the *fcntl* system call which can be used to get the file access mode, returns a set of flags describing that mode. The *strace* and *truss* utilities use different names for the same flag. The parser is

```
# strace output
11470 fcntl(4, F_GETFL)    = 0x402 (flags O_RDWR|O_APPEND)

# truss output
2361: fcntl(4, F_GETFL)    = 10
    FWRITE|FAPPEND

# common representation
('fcntl_syscall', (4, ['F_GETFL']), (['O_APPEND', 'O_RDWR'], None))
```

Figure 11: Renaming flags to provide a consistent uniform representation

```
# strace output
12849 access("syscalls.txt", R_OK|W_OK) = 0

# dtrace output
1128  access("syscalls.txt", 6)         = 0

# common representation
('access_syscall', ('syscalls.txt', ['R_OK', 'W_OK']), (0, None))
```

Figure 12: Translating number to a list of flags

able to detect this discrepancy and amend it, as shown in figure 11. On the other hand, *dtrace* does not use flag names at all, but rather gives a number in place of a flag. This number is made up from the numeric representations of each individual flag, *bitwise ORed* with each other. The parser can detect this too and translate that number into a set of flags, by using the lexical representation of each flag taken from the *Linux header files* [14, 6]. This is demonstrated in figure 12. Similarly, the parser can deal with other cases when values have the same content but different representations, for example timestamp values.

Figure 13 shows an example of the *Str()* special class. This class is used to parse values that are expected to be of string type. Lines 13 to 15 demonstrate that if the value being parsed is not of the expected type, an exception is raised, stopping the parser from proceeding. The developer can then examine the problem that caused this exception and make appropriate adjustments. When such an exception is raised, one of two things must have happened. Either the parser has a bug that needs to be fixed or the trace being parsed includes a value with a format that the parser is not aware it can exist. Lines 2 and 3 show the constructor method of this class. The constructor can optionally take a boolean argument called "output". How this argument is used is explained in section . Lines 6 to 11 show how the parser deals with the fact that the value might be missing (*val == None*) or the value might not have been dereferenced (*val.startswith('0x')*). These situations are further explained in section 5.4. Finally, lines 17 to 20 show some format translations performed on this type of values.

```
class Str():
  def __init__(self, output=False):
    self.output = output

  def parse(self, val=None):
    if val == None:
      return Unknown('Str', val)

    # if the value starts with 0x it was not dereferenced.
    if val.startswith('0x'):
      return Unknown('Str', val)

    # if the format is not the expected one, raise an exception.
    if not (type(val) is str or type(val) is unicode):
      raise Exception("Unexpected format: " + val)

    # replace special characters.
    val = val.replace("\\n", "\n")
    # ...
    val = val.replace("\\0", "\0")

    return val
```

Figure 13: Part of the *Str()* class used to parse and define objects of string type.

5.4 Handling missing and unexpected values

In some situations, the value being parsed might be missing or not be of the expected type, but the parser can be aware that this situation can happen. Such a situation can appear for example when an error occurs in a system call, in which case strings and structures are not dereferenced. In this case, the trace will include the address of the string or structure rather than their actual values. If such a vase occurs, instead of raising an exception, the parser marks that value as an *Unknown* object. Therefore, before performing any future operations on a value, the type of that value can first be examined for whether it is an *Unknown* object. To avoid any false comparisons, the equality and inequality operators of the Unknown object were overridden, as this is shown in figure 14. By using the *Unknown* object, the parser can deal with the fact that some values are missing. This way, the system call can still be parsed into an action and even if some values are missing, the action can still be used to describe the behavior of the *POSIX API*.

5.5 Forming the uniform representation

The return value of all system calls is an integer value, not necessarily of type int though. In order for a system call to pass additional values to its caller, the caller must allocate some memory before calling the system call, and its address should be passed to the system call as one of its arguments. For example, if the system call needs to return a structure, then that structure needs to be initialized and a pointer to its memory location needs to be passed to the system call. If the system call is successful (no errors occur), then the values that need to be returned will be stored in the memory location pointed to by the address passed to the system call.

```
class Unknown():
  def __init__(self, t=None, v=None):
    self.expected_type = t
    self.given_value = v

  # override equality
  def __eq__(self, other):
    return type(other) is type(self)

  # override inequality
  def __ne__(self, other):
    return not self.__eq__(other)
```

Figure 14: The unknown object used to mark the missing values of a system call.

```
 1  # parsing specification of read
 2  "read": {
 3    'args': (Int(), Str(output=True), Int()),
 4    'return': (Int(), NoneOrStr())
 5  },
 6
 7  # strace output
 8  28433 read(3, "data", 32768) = 4
 9
10  # common representation
11  ('read_syscall', (3, 32768), (4, ('data',)))
```

Figure 15: Rearranging the system call arguments to accurately capture the behavior of the *POSIX API*. The figure demonstrates how the “data” text was moved from the arguments in the system call to the return part in the common representation. This was done based on the parsing specification of the *read* system call (lines 1 to 5)

When tracing a system call that needs to return more than an integer value, the output format of the traced system call is somewhat tangled. This is true for the output of all tracing utilities used (*strace*, truss and dtrace). Specifically, all the returned values are displayed in the trace output as if they were arguments passed in the the system call. This is demonstrated by the read system call in line 2 of figure 3. Even though the text read from the file is the return value of the read system call, it is displayed as if it is one of its arguments. In order to correctly, capture the behavior of a system call, the uniform representation had to be formed in a way that clearly distinguished the arguments from the return values of a system call. This is achieved by specifying in the parsing specification which arguments of an action are indeed arguments passed to the system call, and which are return values.

Figure 15 shows the parsing specification of the read system call. The second argument of read is of the *Str()* class and it is used to hold the text read from the file. The *Str()* class, and all the other parsing classes, can take an optional argument called “output”. If this argument is set to *True*, it indicates that the current value is not an argument passed to the system call but rather one of its return values. Using this information, the arguments of an action can be rearranged so that the resulting uniform representation accurately represents the behavior of the system call. As shown in this figure, even though the *strace* output displays the read text as an argument of the *read* system call, in the uniform representation, this text is moved to the return part.

In some cases, an argument is used for both passing some value to the system call, and returning a value from the system call. These arguments are called *value-result* arguments [9]. An example of such argument is the *optlen* argument of the *getsockopt* system call, whose definition is given below.

*int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen);*

The *getsockopt* system call is used to query a socket (*sockfd*) about its options. *optval* is a buffer in which these options will be stored once the system call is executed. The *optlen* argument is initially indicating the size of the *optval* buffer initialized to hold the options that *getsockopt* will return. Once *getsockopt* returns, the *optlen* argument indicates the actual size of the options returned. When a system call containing a *value-result* argument is traced, the resulting trace file contains only the final value of the *value-result* argument. In other words, the initial value of a *value-result* argument cannot be inferred from the trace. *dtrace* can be set to record this initial value by capturing and storing this value in the entry probe, and subsequently passing it to the return probe where it can be printed into the trace file. Unfortunately, *strace* and *truss* do not have this ability. In order to keep a consistent and uniform representation, which was one of the main goals of *POT*, *dtrace* was also set to record only the return value of all *value-result* arguments, hence capturing exactly the same information that *strace* and *truss* do.

5.6 Intelligent parsing

One of the key strengths of *POT* is the fact that it does not simply break traces into meaningless pieces of text but it intelligently extracts pieces of information from the traces and uses them to form a meaningful uniform representation. Each argument and each return value is associated with a special class that indicates its type. What this means is that one could easily search for values of a specific type within an entire trace file (for example file descriptors, flags, ports, IP addresses, etc) effortlessly. Other parsers would have to tediously query each value of each system call for a specific format, in order to achieve this functionality. This is particularly important for system calls with a varying set of arguments. For example, system calls that take a *sockaddr* structure as one of their arguments, can have an IP and a port arguments or they can have a pid and a groups arguments instead. This can be trivially detected in POT.

5.7 Automatic content-based adaptation

In order to deal with the fact that some system calls can have a varying set of arguments, *POT* extracts the contents of a system call progressively. Based on the contents already read, it automatically adjusts the action specification of the system call being parsed, to

```
 1  # definition of sendto system call
 2  ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
 3                 const struct sockaddr *dest_addr, socklen_t addrlen);
 4
 5  # example of strace output for sendto with AF_INET protocol family
 6  49823 sendto(3, "message", 6, 0, {sa_family=AF_INET, sin_port=htons(25588),
 7               sin_addr=inet_addr("127.0.0.1")}, 16) = 6
 8
 9  # initial action specification
10  "sendto": {
11    'arguments': (Int(), Str(), Int(), Flags(), Flags(), Int(), Str(), Int()),
12    'return': (Int(), NoneOrStr()) }
13
14  # common representation
15  ('sendto_syscall', (3, 'messge', 6, 0, ['AF_INET'], 25588, '127.0.0.1', 16),
16                     (6, None))
17
18  # example of strace output for sendto with AF_NETLINK protocol family
19  28721 sendto(4, "messge", 6, 0, {sa_family=AF_NETLINK, pid=0,
20               groups=00000000}, 12) = 6
21
22  # automatically adapted action specification
23  "sendto": {
24    'arguments': (Int(), Str(), Int(), Flags(), Flags(), Int(), Int(), Int()),
25    'return': (Int(), NoneOrStr()) }
26
27  # common representation
28  ('sendto_syscall', (4, 'messge', 6, 0, ['AF_NETLINK'], 0, 0, 12), (6, None))
```

Figure 16: Example of automatic content-based adaptation. The arguments of the action specification of the *sendto* system call are automatically adjusted.

reflect the correct expected arguments. Detecting situations when this is necessary can be achieved very easily since finding arguments of specific type is a trivial job in *POT*.

Figure 16 demonstrates a situation when this can happen. Lines 1 to 3 give the definition of the *sendto* system call. *sendto* is used to send messages to other sockets. As part of its arguments it takes a sockaddr structure which contains a flag indicating the socket family of the sockaddr structure which is used to specify the type of communication to be carried. If this flag is set to *AF_INET* then the sockaddr structure will contain the IP and the port needed to carry out an IPv4 communication. If this flag is set to *AF_NETLINK* though, it indicates that the communication should be between the kernel and the user process, in which case the sockaddr structure does not need an IP and port but rather the pid and the groups. Lines 9 to 12 show the initial action specification that expects a port number and an IP address (*Int()*, *Str()*) and lines 22 to 25 show the automatically adjusted action specification which requires a pid and a groups value (*Int()*, *Int()*)

6 Storing the environment state.

Once a trace file describing an application's execution is traced and parsed, the information of all system calls involved during that execution are captured and used to form the uniform representation. In some cases, this representation is not enough to fully describe the behavior of the *POSIX API*. This is because the behavior of some system calls rely on state that is predetermined before an application starts its execution. For example, the result of some system calls can be affected by the state of the environment in which the application is executed.

Many system calls are directly affected by the state of the file system. An application's behavior is often dependent on the existence of certain files. For example, the *open* system call will only succeed if the file attempting to open exists. If the file does not exist, *open* will fail (assuming the *O_CREAT* flag was not set) and return an error message. In order to fully capture the behavior of the *POSIX API* during an application's execution, the environment state has to be recorded as well. Once a trace is parsed, it is used to identify what additional information is needed. Specifically, all the system call actions are searched for references to files. Whenever an action having a file path in its arguments is found, that action is examined to see if the file it refers to existed in the local file system the at time the trace was captured. This can be achieved by examining the return value of that system call. If the system call failed with an *ENOENT* error label, then the file did not exist. Otherwise, that file must have existed and it is hence stored along with the parsed trace.

7 Analyzing POSIX API Behavior

Once the behavior of the POSIX API is captured and parsed in a clear and common representation, external applications can be built to make use of this common representation in order to analyze the POSIX behavior. As part of this work, several application plug-ins were developed and used to generate statistics and discover patterns in the POSIX behavior. Building these plug-ins took minimal effort. Because of the clear and easy to use format of the *POT* output, useful plug-ins can be built within a few hours. Furthermore, because of the generic nature of the uniform representation, these plug-ins can be effortlessly ported to many platforms.

Because of the wide range of system calls *POT* supports, there are many areas in which it can be used. Firstly, a set of plug-ins are explained, which make use of *POT* to provide I/O statistics and I/O support. Then *POT* is used as the base for a plug-in used to provide network support. Finally, the usefulness of *POT* is demonstrated through a plug-in that is used to to provide security support.

7.1 I/O support

One possible way in which *POT* can be used, is to provide I/O statistics and I/O support. By being able to intercept and parse several I/O related system calls, *POT* is able to provide a uniform representation that can be easily analyzed and used to extract useful I/O information. To demonstrate this ability, two plug-ins were developed.

The first plug-in uses the uniform representation provided by *POT* to generate a set of I/O statistics. As shown in figure 17, this plug-in can generate several different statistics. Firstly, it counts how many times each I/O related system call occurred in the trace. This is useful in getting a general idea on how I/O intensive the traced application was. Then it shows the average amount of data read and written per system call, alongside the average buffer size used in those system calls. Apart from that, a finer analysis is given on the sizes of the data read and written. Lines 15 and 22 show the number of *read* and *write* system calls respectively, according to the size of data read or written. This can be particularly useful when reasoning about the performance of an application. For example if an application performs many reads and writes in small chunks of data, it will most likely have worse performance that if it performed less I/O calls with bigger chunks of data.

Furthermore, the number and type of errors that occurred, in all the *read* and *write* system calls is also provided. If this number is unusually high, it could serve as a warning that the application might be doing something wrong. Finally, this plug-in prints a list of all the files that appear in the trace. Apart from exploring aggregated I/O statistics about the entire trace, it is also very useful to examine the system call activity on specific files. This idea is further explored through another plug-in explained in section 7.3

Inspired from the previous plug-in, another I/O related plug-in was developed to explore the I/O information that *POT* could provide. This plug-in focuses on all the file paths that can be found in the trace file. Using these file paths, the plug-in firstly checks whether the files corresponding to those file paths existed at the time the trace was gathered. This is

```
I/O System call count:
    access: 84
      stat: 14
      read: 188
    statfs: 1
     mkdir: 1
     write: 23
     rmdir: 2
    unlink: 3
      open: 196

- average buffer size per read syscall:  1964
- average data read per read syscall:    1514
Number of reads per size:
<100b:32, <1000b:92, <10000b:64, >=10000b:0
Number of read errors:
     EAGAIN :  4

- average buffer size per write syscall:   12
- average data written per write syscall:  12
Number of writes per size:
<100b:23, <1000b:0, <10000b:0, >=10000b:0
Number of write errors:
    None

File paths:
     /etc/ld.so.nohwcap
     /etc/ld.so.preload
     /etc/ld.so.cache
     /lib/i386-linux-gnu/libc.so.6
     ...
```

Figure 17: I/O statistics generated by analyzing the behavior of *POSIX API*

achieved by examining the return value of the system call the file path was taken from. If the return value indicates that the system call was successful, then the file must have existed. If the system call failed then the error label is further examined to see the type of error that occurred. If for example the error label is *EACCES*, it indicates that the system call failed because it did not have the required permissions to access the file. In this case the file exists, even though the system call failed.

When the file paths are filtered and only the ones referring to an existing file are left, they are appropriately arranged and used to construct a file hierarchy. Figure 18 shows an example of a file hierarchy constructed using a trace gathered during the execution of the *Firefox* browser under Linux and under Solaris. Lines 1 to 9 show a small part of the file hierarchy constructed from the *Linux* trace and lines 11 to 23 the *Solaris* one. Subsequently, these two file hierarchies can be compared, to examine the different files involved in each operating system. Interestingly enough, the figure shows that the trace of *Firefox* on *Linux* has a system call that makes use of the "/etc/passwd" file. In section 7.3 a plug-in is discussed that can be used to provide security support.

7.2 Network Support

Unlike other parsers that only support I/O system calls, *POT* supports system calls of various classifications. As a result, apart from I/O support, *POT* can also provide network support. A plug-in was specifically developed to extract information from the network related system

```
 1  /
 2      /dev
 3          /dev/null
 4          /dev/nvidia0
 5          /dev/nvidiactl
 6      /etc
 7          /etc/ld.so.cache
 8          /etc/nsswitch.conf
 9          /etc/passwd
10
11  /
12      /ICEauthority
13      /Xauthority
14      /dev
15          /dev/tty
16          /dev/udp
17          /dev/urandom
18      /etc
19          /etc/default
20              /etc/default/nss
21          /etc/fonts
22              /etc/fonts/conf.d
23                  /etc/fonts/conf.d/20-fix-globaladvance.conf
```

Figure 18: Part of the file hierarchy showing all the files involved in the interactions between the *Firefox* browser and the *POSIX API system call interface*. Lines 1 to 9 show a small part the file hierarchy constructed using a trace gathered in *Linux* and lines 11 to 23 using a trace gathered in *Solaris*.

calls of a trace file and provide network statistics. Figure 19 shows the output of this plug-in. Lines 1 to 7 show information about the sent data and lines 9 to 15 show information about received data. Firstly the total number of *recv* or *send* system calls is given. Then an average of the data sent/received per system call is shown followed by the average buffer size per system call. Examining the average buffer size allocated in an application and comparing it with the average data sent or received can be very useful. For example, if the buffer size of the *recv* system calls is constantly significantly higher than the actual data received, then that means that extra memory is unnecessary being reserved. This can potentially hurt the performance of the application. After observing this behavior, adjustments can be made to the application so that the allocated memory matches the needs of the application. Another information provided by this plug-in is which socket address families were used to carry out the communications. This information can be used for example to check whether applications use IPv4 or IPv6.

7.3 Security Support

POT can also be used as the base for applications that aim to provide security support. In order to demonstrate this, a final plug-in was developed. The purpose of this plug-in is to track all the actions performed on specific files. To achieve this, the plug-in must deal with two kinds of system calls. Firstly, it must find all the system calls that contain a file path. This can be achieved very easily. Because of the fact that *POT* does not simply parse traces into a sequence of strings but rather forms actions containing typed arguments, each action can be trivially queried for whether it contains a file path. The second kind of system

```
Total number of send syscalls:        4
Average buffer size per send syscall:  14
Average data sent per send syscall:    14
Number of socket families used:
     PF_INET : 4
Number of send errors:
    None

Total number of recv syscalls:          25842
Average buffer size per recv syscall:   4162
Average data received per recv syscall: 100
Number of socket families used:
     PF_INET : 25842
Number of recv errors:
     EAGAIN :  51654
```

Figure 19: Network statistics generated by analyzing the behavior of *POSIX API*

calls that this plug-in needs to deal with, are the system calls that contain a file descriptor instead of a file path. This case is not as trivial. In order for the plug-in to extract useful information from system calls that contain a file descriptor, it needs to keep track of all file paths met and associate them with the correct file descriptor. This is achieved by adding a dictionary entry whenever an *open* system call is found, that associates the file path with the resulting file descriptor. Similarly, when a *close* system call is met, the corresponding file path to file descriptor entry is removed from the dictionary. The current implementation of *POT* does not support this functionality by default, hence this had to be explicitly dealt with within the plug-in. This functionality is further discussed in future work in section 9.

Figure 20 shows a sample output of the plug-in discussed above. This is a small part of the output taken by using a trace file that contained the POSIX API behavior of the *useradd* command. For each file existing in the trace, a summary with all the system calls involving that file is given. The plug-in can also be used to track the actions of a specific file as well. The first system call tracked is usually the *open* system call. This is shown in lines 1, 10 and 19 of the figure. Below every *open* system call, there is a set of system calls involving that file. For example, the first file tracked in the figure is the "/etc/default/useradd" file. On this file the system calls *fstat*, *read* and *close* are performed.

A possible way to make use of these results is to check for potential security threats. To further support this use, the plug-in marks some system calls whose execution can potentially be a threat to the system. Examples of these system calls are *chmod* and *link*. *chmod* could be used to change the permissions of a system file in order to compromise the system. For example if an application executes the system call *chmod("/etc/passwd", 0666)* it should definitely raise some suspicion. Similarly, an application could try something like this:

unlink("/etc/passwd")
link("/tmp/passwd","/etc/passwd")

These cases could be easily discovered through the output of this plug-in.

```
OPEN "/etc/default/useradd" flags: O_RDONLY|O_LARGEFILE
     fstat
         arguments: 3,
         return: 0, 'makedev(8, 6)', 392963, ['S_IFREG', 'S_IWUSR', ...
     **READ
         arguments: 3, 4096
         return: 1118, '# Default values for useradd(8)\n#\n# The ...'
     CLOSE FILE

OPEN "/etc/passwd" flags: O_RDONLY|O_CLOEXEC
     fstat
         arguments: 4,
         return: 0, 'makedev(8, 6)', 452759, ['S_IFREG', 'S_IWUSR', ...'
     **READ
         arguments: 4, 4096
         return: 1914, 'root:x:0:0:root:/root:/bin/bash\ndaemon:x: ...'
     CLOSE FILE

OPEN "/etc/passwd.374" flags: O_WRONLY|O_CREAT|O_EXCL|O_LARGEFILE
     write
         arguments: 4, '374\x00', 4
         return: 4, None
     CLOSE FILE
     **LINK
         arguments: '/etc/passwd.374', '/etc/passwd.lock'
         return: 0, None
     **UNLINK
         arguments: '/etc/passwd.374',
         return: 0, None
```

Figure 20: Part of the output of the plug-in used for tracking all activities per file, for all files involved in the *useradd* command

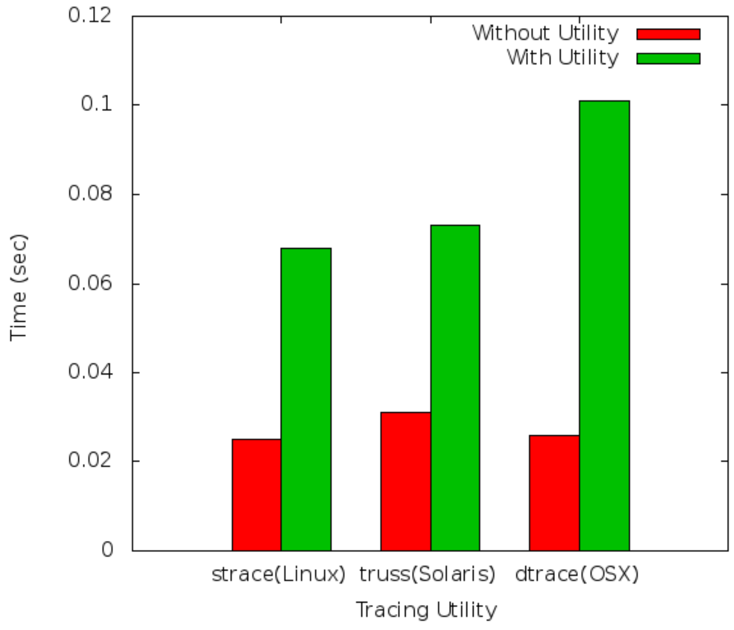

Figure 21: Time comparison of running an application with and without a tracing utility

8 Evaluation

8.1 Performance

In order to determine the total performance impact of *POT*, firstly the performance of the trace capturing mechanism had to be evaluated. With this goal in mind, a simple test program was developed that would run the set of the 55 supported system calls hundreds of times within a loop. This program was then executed under different platforms. In each platform, two kinds of measurements would be made. Firstly, the performance of the test program itself would be measured. Then this program would be traced using the available tracing utility on that platform and the time taken for the entire procedure to complete would be measured. Figure 21 shows a plot with the resulting measurements for each tracing utility. The difference between the time taken for the program to execute and the time taken for the tracing utility to trace this program is significant. This performance drawback is likely being caused because of the high number of context switches that have to be made, in order for the tracing utilities to be able to capture the behavior of the kernel system calls. *dtrace* in particular, affected the performance of the program heavily. This is related to the fact that the *d scripts* used to capture the behavior of the *POSIX API*, involve a lot of *dtrace* probes which can affect the performance of the *dtrace* utility.

Once the performance of the tracing utilities was evaluated, the performance of *POT* had to be tested. To achieve this, *POT* was used to parse trace files of various sizes. Figure 22 shows a plot of the recorded times taken to parse the behavior of trace files starting from a thousand lines long up to a 160 thousand lines long. Because *POT* is an I/O intensive tool, its performance was compared with that of the performance of the *grep* tool. The performance

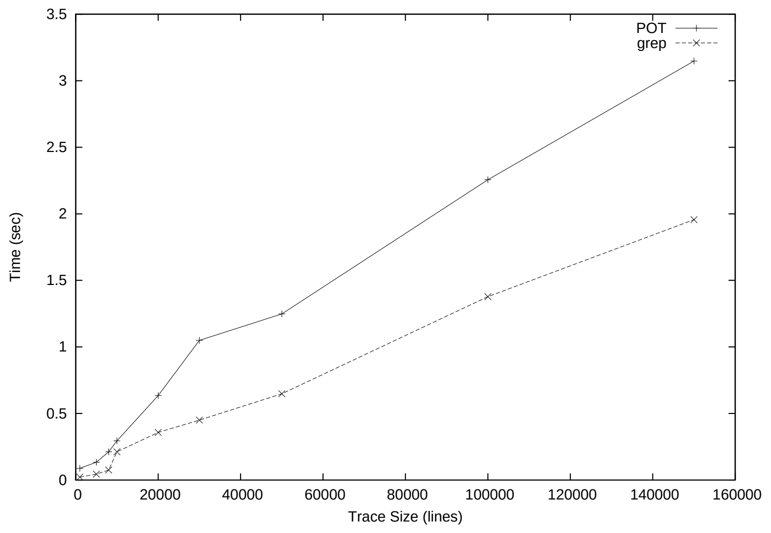

Figure 22: Time comparison between *POT* and the *grep* utility.

of *POT* is very close to the performance of *grep* for trace files up to 20 thousand lines. For bigger files, *POT* is a little slower. This was expected because for each line of the trace file being parsed, *POT* has to do a lot more work than *grep*. *POT* has to break the system call into its parts, extract all the arguments and return values of the system call, assign a type to each of these values and reform the system call into a uniform representation. In addition, *POT* needs to keep some state about the system calls already met, which explains why *POT*'s performance is a little worse than the performance of *grep*.

8.2 Usability

Before *POT* could be used to support real applications, it was necessary to confirm that the tracing and parsing mechanisms were working accurately. To do that, a way had to be found to generate simplistic trace files, to test *POT* with. This was achieved by developing a *C* program and use it to simulate the execution of individual system calls. This program would use the system call wrappers provided by the *glibc* library to invoke the kernel system calls. Subsequently, the execution of this program would be traced using strace, truss and dtrace to generate a simple trace file for each of the system calls supported by *POT*. Once these traces were generated, *POT* was used to parse them into the common representation. Then the common representation was compared with the definitions of each system call to confirm that the format and the content of all actions in the common representation were correct.

Figure 23 shows two snippets from the *C* program used to simulate the execution of system calls. Lines 1 to 16 show the function used to simulate the *lseek* system call. *lseek* is used to change the offset in a file so that future operations on that file (eg *read* and *write*) will be performed starting at the new position pointed to by the offset. In this function the

```
 1  int execute_lseek() {
 2    // create a file and write a sample text in it.
 3    int fd = open("file.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR|S_IRGRP);
 4    write(fd, "abcdefghijklmnopqrstuvwxyz", 26);
 5    char buffer[10];
 6
 7    // set the offset to the second character and read 10 characters.
 8    lseek(fd, 1, SEEK_SET);
 9    read(fd, buffer, 10);
10
11    // set the offeset to 5 characters after the current offset and read.
12    lseek(fd, 5, SEEK_CUR);
13    read(fd, buffer, 10);
14
15    return 0;
16  }
17
18  int execute_getdents() {
19    // initialize the buffer and open the current directory.
20    int fd, nread, BUF_SIZE = 1024;
21    char buf[BUF_SIZE];
22    fd = open(".", O_RDONLY | O_DIRECTORY);
23
24    for ( ; ; ) {
25      nread = syscall(SYS_getdents, fd, buf, BUF_SIZE);
26
27      if (nread == -1)
28        break;
29      if (nread == 0)
30        break;
31    }
32    return 0;
33  }
```

Figure 23: Simulating the behavior of system calls by using *glib* system call wrappers to invoke the system calls.

lseek system call wrapper is used twice, with different values each time, in each case followed by the *read* system call wrapper to confirm that the *lseek* system call was executed correctly. In lines 18 to 33 the function for simulating the *getdents* system call is given. This function is different than the previous one because *glibc* does not provide a wrapper for the *getdents* system call. So in order to invoke *getdents*, the *syscall* function, which is used to indirectly call *getdents*.

Once the tracing and parsing mechanisms of *POT* were confirmed to work correctly, *POT* was used as part of a bigger system called *CheckAPI*. *CheckAPI* is a tool used for testing for portability violations of portable APIs. It examines the interactions of an application with an API (in this case the POSIX API) and verifies that the observed behavior matches the specification of the API. Specifically, it makes use of the actions extracted from a trace file using *POT* to simulate the expected API behavior and subsequently, discover discrepancies between the recorded execution of the API and the simulated execution of the API. With the help of *POT*, *CheckAPI* successfully discovered a case were a difference in the behavior of the *POSIX API* on different operating systems caused a portability violation. Specifically, the behavior of the *accept* system call on Linux systems is different that the one on *BSD* systems. On Linux, when a new socket is returned by the *accept* system call, it does not inherit the O_NONBLOCK and O_ASYNC flags from its parent socket. In contrast, on

BSD systems, these flags are inherited to the new socket returned by the *accept* system call. This behavioral difference was causing the *Python* programming language running on a *BSD* operating system to raise an error under certain conditions [15].

9 Conclusion and Future Work

In this thesis the *POSIX Omni Tracer* tool was described. A way to trace the *POSIX API* behavior was explained, that is applicable in a variety of platforms. A uniform common representation was also introduced, which is able to describe the behavior of the *POSIX API* captured in any of these platforms. Finally, the usefulness and applicability of *POT* was demonstrated through various plug-ins but also through applications of bigger scale.

There are a few areas in which *POT* could be expanded. As previously stated, the current implementation of *POT* supports the 55 most frequently used system calls. In the future *POT* could be expanded to support more system calls so that it can capture a more accurate representation of the *POSIX API* behavior. Apart from adding support for more system calls, *POT* can also be improved in parsing more information for each system call. For example, the elapsed time for each system call can be recorded, which could be used to generate more statistical information for an application. Another way in which *POT* can be expanded is to provide support for multi- threaded applications. At this point, *POT* does not record the *pid* of each system call. This means that it cannot differentiate system calls executed in the main thread from system calls executed in child notes. Many useful plug-ins could make use of this to provide analytics. Finally, *POT* can also be expanded to support more platforms. *Cygwin* for example is a POSIX-compliant environment for Microsoft Windows system that *POT* could potentially provide support for. Additionally, *Interix* is another Microsoft Windows POSIX-compliant subsystem in which *POT* could be used.

References

[1] B2G Architecture. Mozilla Wiki, https://wiki.mozilla.org/B2G/Architecture, Mar 2013.

[2] Pystrace tool. https://code.google.com/p/pystrace/, Apr 2013.

[3] STrace ANAlyzer (stana) tool. https://github.com/johnlcf/Stana/wiki, Apr 2013.

[4] The dtrace utility. https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/dtrace.1.html, Mar 2013.

[5] The dtruss utility. http://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/dtruss.1m.html, Mar 2013.

[6] The fcntl header file. http://linux.die.net/include/fcntl.h, Feb 2013.

[7] The IOapps suite. http://code.google.com/p/ioapps/, Apr 2013.

[8] The ktrace utility. http://www.manpagez.com/man/1/ktrace/, Mar 2013.

[9] The manual page for the getsockopt system call. http://linux.die.net/man/2/getsockopt, Feb 2013.

[10] The ptrace system call. http://linux.die.net/man/2/ptrace, Mar 2013.

[11] The strace analyzer. Jeff Layton http://en.community.dell.com/techcenter/high-performance-computing/w/wiki/2264.aspx, Apr 2013.

[12] The strace utility. http://linux.die.net/man/1/strace, Mar 2013.

[13] The truss utility. http://docs.oracle.com/cd/E19082-01/819-2239/truss-1/index.html, Mar 2013.

[14] The unistd header file. http://linux.die.net/include/unistd.h, Feb 2013.

[15] On mac / bsd sockets returned by accept inherit the parent's fd flags. Accessed May 2013 http://bugs.python.org/issue7995.

[16] Griffin, M. Firefox OS. Mozilla Developer Network, https://developer.mozilla.org/en-US/docs/Mozilla/FirefoxOS, Apr 2013.

[17] N. Gandhewar1, R. S. Google Android: An Emerging Software Platform For Mobile Devices, 2012.

[18] Pichai, S. Introducing the Google Chrome OS,. The Official Google Blog, http://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html, Jul 2007.

[19] Vaughan-Nicholas, S. Ubuntu for android: Linux desktop on a smartphone. http://www.zdnet.com/blog/open-source/ubuntu-for-android-linux-desktop-on-a-smartphone/10402, Feb 2012.