Czech Technical University in Prague

Faculty of Information Technology

Department of Theoretical Computer Science

Master's thesis

Port of Valgrind to Solaris/x86

Bc. Petr Pavlů

Supervisor: Mgr. Jiří Svoboda

9th May 2012

Acknowledgements

I would like to thank my supervisor Jiří Svoboda for providing me help and very good advices, and my family for supporting me throughout my studies.

Declaration

In Prague 9th May 2012 ....

Citation of this thesis

Petr Pavlů. *Port of Valgrind to Solaris/x86: Master's thesis.* Czech Republic: Czech Technical University in Prague, Faculty of Information Technology, 2012.

Abstract

The aim of this thesis is to port Valgrind to the Solaris/x86 platform. Valgrind is a combination of a dynamic binary analysis framework and associated tools. It is a free application licensed under the GNU GPL 2. The tools are able to find many programming errors, for example, memory leaks, race conditions or underperforming code. Solaris is a commercial Unix-type operating system developed by the Oracle Corporation. The thesis also describes main directions and techniques in the program analysis area, together with examples of actual tools.

Keywords dynamic program analysis, Valgrind, Solaris

Abstrakt

Cílem této diplomové práce je vytvoření portu nástroje Valgrind na platformu Solaris/x86. Valgrind je kombinace frameworku pro dynamickou binární analýzu a k tomu přidružených nástrojů. Jedná se o svobodnou aplikaci licencovanou pod GNU GPL 2. Nástroje umožňují uživatelům najít mnoho chyb v jejich programech, např. úniky paměti, souběhy nebo nedostatečně optimalizované části kódu. Solaris je komerční operační systém unixového typu vyvíjený společností Oracle. Text práce také obsahuje popis hlavních směrů a technik v oblasti analýzy programů společně s příklady konkrétních nástrojů.

Klíčová slova dynamická analýza programů, Valgrind, Solaris

Contents

Figures

Tables

Listings

CHAPTER 1

Introduction

No software product is bug free. Finding bugs and fixing them form a big expense in the life of any nontrivial program. Special tools have been developed to simplify this task. Some of the most used ones are profilers, static analysers and dynamic analysers. Together they are called program analysis tools.

The Solaris operating system provides many such programs. They allow to easily inspect what is happening inside the kernel or in user programs. In addition to common Unix tools, Solaris also contains its own specific program analysis tools, for example, libumem to detect common memory related errors and dynamic tracing facilities provided by the DTrace tool. However, an easily accessible heavyweight dynamic analysis framework is not available.

Valgrind is a combination of a dynamic binary analysis framework and associated tools. It uses a disassemble-and-resynthesise technique that provides great control over a running program but in exchange, it requires extra system resources and significantly slows down the observed program (typically 10-50 times slower than the native execution). Valgrind is a free application licensed under the GNU GPL 2. The official version currently supports Linux and Mac OS X (an unofficial port to FreeBSD also exists). It can run x86, AMD64, PPC32, PPC64 or ARM binaries. Three categories of tools are available. Memory error detectors are able to find memory leaks, use of uninitialised memory, accesses to freed memory and out-of-bounds memory accesses. Thread error detectors are capable of recognising race conditions in multi-threaded code. The last category consists of cache and heap profilers, they help to make programs run faster. The most used tool is from the first category and is called Memcheck.

1.1 Goals

The main goal of this thesis is to port Valgrind to the Solaris/x86 platform (or more specifically to the SunOS 5.11 kernel). This task is not as trivial as porting a usual user program between POSIX-compliant systems because Valgrind is closely tied to details of an operating system. This thesis consists of three general goals:

- Study program analysis tools and compare their basic capabilities.
- Study Valgrind internals, especially those depending on the underlying operating system. Devise a way of porting Valgrind to the Solaris/x86 platform and implement it.
- Demonstrate the functionality of the port by running Valgrind's Memcheck tool on a simple command-line application. Optionally, verify the functionality of the tool using the regression tests bundled with the Valgrind source code.

Chapter 2

Program Analysis

2.1 Purpose of Program Analysis

Program analysis is a process of automatically analysing a behaviour of computer programs. The intention is to find non-obvious properties of observed programs that expose optimisation opportunities or manifest bugs in the programs. The origin of program analysis is dated to the advent of the first compilers where it was used to optimise compiled programs. That started a complex development in this field, essentially becoming a wide research and industrial area. Besides program analysis embedded in compilers and similar systems, there are now also many different specialised tools directly targeted at programmers that help them to deal with increasingly complex computer systems.

Main applications of program analysis:

- *Program optimisation*, for making code run faster and use less memory. This ranges from simple constant propagation optimisations used in compilers to heap or cache profiling.
- *Program correctness*, for detecting bugs in programs before they land on production systems. This includes finding memory leaks, invalid pointer references, use of undefined values, or race conditions.

Program analysis can be split into two categories: static analysis and dynamic analysis. The next pages describe these two concepts. Since Valgrind is a dynamic analysis tool, the main focus is on dynamic analysis and static analysis is discussed only marginally. The following description also mentions several representatives of different kinds of analysis methods. The attention is given especially to tools that are not parts of typical compiler suites and to tools that are available on the Solaris operating system.

2.2 Static Analysis

2.2.1 Basic Description

Static analysis is a name for methods that derive properties of a program without executing it. The methods are based on examining a source code of the program (typical for machine code compiled languages, such as C/C++) or a compiled code (typical for bytecode compiled languages, such as Java). Compiler optimisations are standard examples of static analysis.

Bug searching tools based on this approach usually used to be sound. The soundness property guarantees that a tool produces a description of a program's behaviour which holds for all possible inputs and runtime environments. Unfortunately soundness usually comes at the cost of less precise results, with false positives. If there are too many false positives in a tool's report then the tool becomes unusable. Reducing a number of these warnings in an error-free code is a key challenge for all static analysis tools. In order to do so, some modern tools do not hold the soundness property [1].

The tools operate by creating a model of a program state. Because there are many possible executions, many possible states must be tracked. However, in practise, tracking all possible precise states cannot be done because it would require too much (possibly unbounded) computing time and memory. Thus the static analysis tools must use some approximation, resulting in an information loss. Even if there is enough computing time and memory available, the approximation is still inevitable as many facts of a program's behaviour are undecidable, for example, deciding whether `exit(2)` is ever called (the halting problem).

Another complication, that the static analysis tools have to deal with, are undefined values. Such values come from non-deterministic inputs, for example, from a user or a network. In order to preserve soundness, analysing has to be conservative and must consider all possible values from such inputs. If a tool operates at the source code level then there is also a problem that not all sources can be accessible and so semantics of these functions is not directly available. Such obstacles can considerably degenerate a tool's output.

2.2.2 Implementation Techniques

Static analysis methods are usually based on rigorous mathematical approaches. There are several of them but unfortunately this thesis does not have room to describe many of them. Two very common, which also nicely denote basic ideas of static analysis, are: a data flow analysis and an abstract interpretation.

2.2.2.1 Data Flow Analysis

The data flow analysis is a method that determines how variables are used within programs. It utilises a control flow graph of a program to obtain required properties. Its main use is in compilers when optimising programs. A typical example of this technique is a reaching definition analysis that calculates which variable definitions can reach a given point in a program's code. The data flow analysis is generally performed by setting up data flow descriptions for each node of the control flow graph and then propagating the descriptions along the paths in the graph.

2.2.2.2 Abstract Interpretation

The abstract interpretation is a theory for obtaining information about a program's behaviour based on an approximation of program semantics. It replaces concrete states in the program by their more general (abstract) form and provides instructions to manipulate this form. For example, an abstraction of each integer variable x can be an interval $I_x = [L_x, H_x]$, where $x \in I$. Using this concrete abstraction, a potential statement $y = u + v$ is calculated as $I_y = [L_u + L_v, H_u + H_v]$. It can be viewed as a partial execution of a program in its abstract form.

2.2.3 Representative Tools

Just as there are many static analysis methods, there are also many tools that use them. Wikipedia, the free encyclopedia, contains a comprehensive list of the static analysis tools [41]. Not to fill many pages with their description, only two notable ones are introduced in the following text: Lint and Coverity Static Analysis.

2.2.3.1 Lint

Lint was the first widely used static analysis tool. It was developed by Stephen C. Johnson in 1977 [13] and first released to the public in 1979 as a part of Unix V7. (Lint's original source code is now available online [60].) It is a command which examines C source programs, detecting a number of bugs and obscurities. It is based on the Portable C Compiler (written by the same author).

A shortened list of issues that Lint detects (nowadays, most of them are discovered directly by compilers):

- unused variables and functions.
- used before set variables,
- an unreachable (dead) code,
- a misuse of function return values (for example, using a function value when the function does not return one),
- a non-portable character use (if the char data type is used with an assumption that it is a signed/unsigned type),
- assignments of longs to ints (possible accuracy loss),
- constructs without any effect,
- multiple uses and side effects (problems such as `a[i] = b[i++];`).

To reduce a number of unwanted warnings, Lint allows to include additional semantic information in a program's code. Such information is written inside comment blocks. Examples are `/*NOTREACHED*/` – indicates that a code after the current line is unreachable, `/*ARGUSED*/` – suppresses a warning about unused arguments for the current function. Modern static analysis tools provide more sophisticated mechanisms how to write extra semantic information but the principle remains the same.

Reincarnations and extended versions of the original Lint can be found on today's systems, for example, PC-lint [47] or Splint [7, 51]. Listing 2.1 shows several defects that can be found by Splint. The example is self-explanatory but note that none of the problems found by Splint is detected by the GNU C compiler.

Listing 2.1: Splint detecting a memory leak and a few other problems

```
setup@sol:~$ cat lintme.c
#include <stdlib.h>
int main(void)
{
  char *i = malloc(10);
  i[12] = 'a';
  return 0;
}
setup@sol:~$ cc -Wall -Wextra -pedantic lintme.c
setup@sol:~$ splint -strict lintme.c
Splint 3.1.2 --- 09 Feb 2011

lintme.c: (in function main)
lintme.c:4:20: Function malloc expects arg 1 to be size_t
  gets int: 10
  To allow arbitrary integral types to match any integral
  type, use +matchanyintegral.
lintme.c:5:3: Index of possibly null pointer i: i
  A possibly null pointer is dereferenced.  Value is either
  the result of a function which may return null (in which
  case, code should check it is not null), or a global,
  parameter or structure field declared with the null
  qualifier. (Use -nullderef to inhibit warning)
   lintme.c:4:13: Storage i may become null
lintme.c:6:12: Fresh storage i not released before return
  A memory leak has been detected. Storage allocated locally
  is not released before the last reference to it is lost.
  (Use -mustfreefresh to inhibit warning)
   lintme.c:4:24: Fresh storage i created
lintme.c:5:3: Likely out-of-bounds store: i[12]
    Unable to resolve constraint:
    requires 9 >= 12
     needed to satisfy precondition:
    requires maxSet(i @ lintme.c:5:3) >= 12
  A memory write may write to an address beyond the allocated
  buffer. (Use -likelyboundswrite to inhibit warning)

Finished checking --- 5 code warnings
```

2.2.3.2 Coverity Static Analysis

In the year 2002, the Coverity, Inc. company was founded by Stanford University computer scientists. The goal of this company was to commercialise a static bug-finding tool that was developed in a four-year research project at the university [6, 9]. The product is today called Coverity Static Analysis and it is one of the most advanced static analysis tools on the market.

It is an unsound tool that can analyse C/C++ at the source code level and Java and C# at the bytecode level. The product aims for less than 20 % of false positives [1].

Analysis used is interprocedural (crosses function boundaries). It means that built-in checkers look at all functions in the context. This technique allows to find more code defects. A patented statistical approach is used in the error reporting, for example, if a function return value is always checked by a caller but there is one place where it is not, then this place is marked as an error. Other features are parallel analysis and incremental analysis (if there is a code change then only affected files are reanalysed) [30].

The following list enumerates defects found using Coverity Static Analysis (carbon copied from the data sheet of the tool [29]):

- concurrency defects such as deadlocks, race conditions and blocking misuse,
- performance degradation problems due to memory leaks, file handle leaks, custom memory and network resource leaks, database connection leaks,
- crash causing errors such as null pointer dereference, double-free, use-after-free, improper memory allocation, and mismatched array new and delete,
- an incorrect program behaviour caused by a dead code, uninitialised variables, invalid use of negative variables,
- improper use of APIs with C++ STL usage errors,
- security vulnerabilities due to buffer overflows, insufficient validation, etc.

In the year 2006, Coverity, Inc. started to scan source codes of lead open source projects. This scan project was founded by United States Department of Homeland Security. In the first year, over 6,000 defects found by the scan were fixed. In the following years the project was extended to include more open source programs. Since 2009, the project is completely founded by Coverity, Inc. [28].

2.3 Dynamic Analysis

2.3.1 Basic Description

An opposite approach to obtain a program's behaviour is used by dynamic analysis, it observes a program by executing it (usually at the machine code level). Typical dynamic analysis tools are profilers and memory leak detectors.

2.3.2 Implementation Techniques

There are several very diverse dynamic analysis techniques, for example, a use of specialised hardware to monitor an execution of a program or a sampling of a program counter for a statistic-based profiling. The next pages describe the most used dynamic analysis method called instrumentation. It is a technique for inserting extra code into a program to observe its behaviour.

The program instrumentation can be divided into three categories: a source-code instrumentation, a static binary instrumentation and a dynamic (runtime) binary instrumentation. There exist a few instrumentation tools from the first two categories, such as Patil and Fischer's bound checker for C [19] (a source code instrumentation tool), or ATOM [22], Purify [10, 40] and FIT [5, 35] (all three static binary instrumentation tools). However, most current tools (that are still actively developed) use the dynamic binary instrumentation. This method has several advantages:

- no need for access to a source code,
- no recompilation of a program is required,
- no files on a disk are changed,
- a dynamically generated code is correctly instrumented,
- some tools also allow to attach to an already running process.

The simplest form of this technique is represented by a function interception via preloading. More advanced approaches are a probe-based instrumentation and a JIT-based instrumentation (copy-and-annotate or disassemble-and-resynthesise).

Note that the following text use two terms: instrumentation code and analysis code. The instrumentation routines select where the instrumentation is inserted and the analysis procedures determine what to do when the instrumentation is activated.

2.3.2.1 Function Interception via Preloading

The principle of the function interception is to redirect one function to another function. Whenever the original function is called, the replaced one is entered. Typical usage involves redirecting a family of the standard memory allocation functions.

There are several ways how the interception can be accomplished. The simplest approach takes advantage of the preload capability of runtime linkers. Preloading allows to insert an arbitrary dynamic library into the symbol resolution process. Symbols in the inserted library have a higher priority than symbols coming from regular libraries required by a program. This permits to effectively intercept all calls to functions contained in dynamically loaded libraries. All hard work is done by the runtime linker, a tool just has to come in the form of a dynamic library. Examples of this technique include libumem (see Section 2.3.3.1), DUMA [32] and mpatrol [42].

2.3.2.2 Probe-based Instrumentation

The probe-based approach allows to insert an analytic code into a specified location in a binary code of a program. The implementation is done by replacing instructions at the location with flow-control instructions that transfer execution to the analytic code.

Figure 2.1 shows an example of two probes inserted into a program. The flow-control is changed using the call instruction. The original instructions are copied to the end of the analysis code. This assumes that the original instructions are not dependent on their location. If it is not the case then such an instruction has to be emulated, executed from the original context or altered in a way that the original behaviour is preserved.

Examples of this technique include Detours [11] (allows to insert probes only at function entries), DTrace (see Section 2.3.3.2) and Pin (see Section 2.3.3.4).

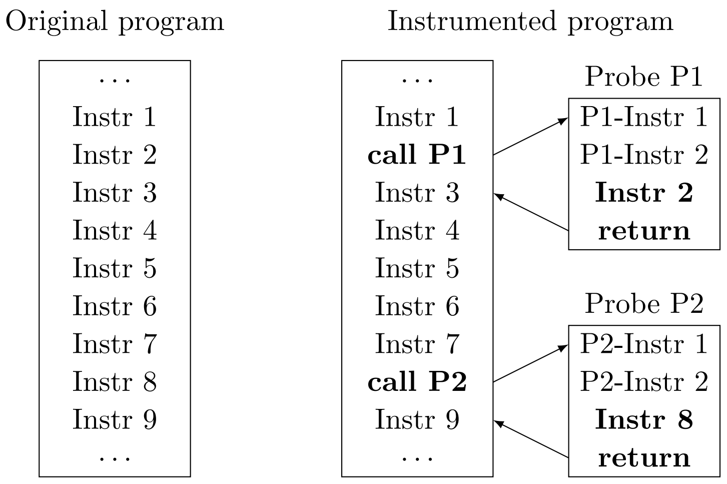

Figure 2.1: Principle of probe-based instrumentation

2.3.2.3 Copy-and-Annotate Instrumentation

The copy-and-annotate instrumentation (C&A) is a runtime technique for inserting extra code into a program that operates on a binary representation of the program and while doing so preserves the original instructions. Only necessary flow control changes are inserted into the program. Each instruction is annotated with a description of its effect, an analysis tool then use the annotations to guide its instrumentation.

The method is said to be transparent which means that it does not change instructions and addresses used in the program. Since the original instructions are (almost) verbatim copied, a complete semantics is retained.

Pin (see Section 2.3.3.4) is probably the most known representative of this approach. It was also used in early versions of Valgrind (see Chapter 3) to handle the FP and SIMD instructions.

2.3.2.4 Diassemble-and-Resynthesise Instrumentation

The disassemble-and-resynthesise instrumentation (D&R) is a similar technique as the copy-and-annotate instrumentation. It also operates at the runtime on a binary representation of a program. However, instead of executing the original instructions, the code is first converted into an intermediate representation (IR). This IR is instrumented (by inserting additional IR) and then compiled back to the original binary format.

The intermediate representation has to be able to represent all instructions of an original code. Simple instructions are directly represented by one primitive operation in the IR, complex instructions are broken into multiple operations and remaining instructions are emulated. For example, if the original binary representation is the x86 machine code then the `cpuid` instruction is a candidate for the emulation.

Compared to C&A, this method is more complex and due to that also slower. D&R is more suitable for heavyweight analysis tools while it is an overkill for simple ones. This method also allows (at least theoretically) a translation from one binary representation to another.

Valgrind (see Chapter 3) is a canonical example of this approach.

2.3.3 Representative Tools

A few dynamic analysis tools were already mentioned in the previous sections. More of them can be found on Wikipedia [34]. This section describes in detail four very popular ones: libumem, DTrace, Oracle Solaris Studio analyser suite and Pin.

2.3.3.1 Libumem

Libumem is a dynamically loaded library consisting of a set of memory allocation functions. It is a port of the kernel SLAB allocator to the user space done in the year 2001 by Jonathan Adams [2, 3]. It is shipped as a part of the Solaris operating system since Solaris 9 Update 3 (released in March 2004). The library is available under the CDDL open source license [43]. There also exists a port to other operating systems [49].

Besides being a highly scalable allocation library [27], it also supports a memory debugging that can be used to detect memory leaks and memory corruption errors. A big advantage is that it is possible to run the libumem memory debugging in the production with an acceptable overhead. The easiest way how to use this library is to utilise the preload facility of runtime linkers (see Section 2.3.2.1).

Listing 2.2 shows an example how libumem is used in a conjunction with MDB[1] to detect a trivial memory leak. In this case, the preload technique is used to inject the library into the program. Setting the environment variable `UMEM_DEBUG` to the `default` value enables the libumem debugging. The program is executed using MDB. A breakpoint is set to the `_exit(2)` call. In a leak-free program, all heap-allocated memory should be freed at this point. The libumem MDB module is loaded to read and interpret the libumem debugging information. Three commands from this module are used in the example. The `::findleaks` command lists all found leaks. The

[1]Modular Debugger (MDB) is an extensible general purpose debugging tool for the Solaris operating system. See the official guide for a complete reference [44].

`::bufctl_audit` command is used to get more detailed information about a specified leak, especially a stack trace indicating where an allocation occurred. Finally, the `::umem_verify` command displays if any allocated buffer is corrupted. Unfortunately, in this case libumem was not able to detect the write after the end of the allocated memory.

Listing 2.2: Using libumem to find a memory leak

```
setup@sol:~$ cat lintme.c
#include <stdlib.h>
int main(void)
{
  char *i = malloc(10);
  i[12] = 'a';
  return 0;
}
setup@sol:~$ cc -o lintme lintme.c
setup@sol:~$ export LD_PRELOAD=libumem.so
setup@sol:~$ export UMEM_DEBUG=default
setup@sol:~$ mdb ./lintme
> ::sysbp _exit
> ::run
mdb: stop on entry to _exit
mdb: target stopped at:
0xfee61508:     nop
> ::load libumem
> ::findleaks
CACHE     LEAKED   BUFCTL  CALLER
08067c10       1 0807be30  main+0x15
-------------------------------------
   Total       1 buffer, 24 bytes
> 0807be30::bufctl_audit
      ADDR     BUFADDR       TIMESTAMP       THREAD
                 CACHE         LASTLOG     CONTENTS
   807be30     8075fd0     55fdbb3d92f            1
               8067c10               0            0
         libumem.so.1`umem_cache_alloc_debug+0x144
         libumem.so.1`umem_cache_alloc+0x19a
         libumem.so.1`umem_alloc+0xcd
         libumem.so.1`malloc+0x2a
         main+0x15
         _start+0x83

> ::umem_verify
Cache Name              Addr      Cache Integrity
[snip]
umem_alloc_24           8067c10   clean
[snip]
```

2.3.3.2 DTrace

DTrace is a dynamic analysis framework developed for the Solaris operating system by Bryan Cantrill, Mike Shapiro and Adam Leventhal [4]. It was first released as a part of Solaris Express 11/03 (released in November 2003) and later shipped in the first version of Solaris 10 (released in January 2005). It is available under the CDDL open source license [43]. This has allowed to create ports to FreeBSD and Mac OS X (ports to several other operating systems are in progress). Besides the official documentation, there also exist two books containing a comprehensive description of this tool [8, 15].

DTrace has been designed to be used on production systems. This ultimately means two things: no overhead when not explicitly enabled and the absolute safety of analysis code execution. It is a kernel based framework, the instrumentation and processing of the analysis code reside in the kernel. This provides an ability to instrument both user and kernel-level software in a unified form. It is done so in a seamless way which allows to follow programs across the user/kernel boundary.

The DTrace architecture consists of three main components: the in-kernel DTrace core, user-level consumers that communicate with the DTrace core via the DTrace library (the `dtrace(1M)` command is a typical consumer) and pluggable instrumentation providers (packaged as kernel modules).

The providers exploit several different types of the probe-based instrumentation (see Section 2.3.2.2). Most probe types have zero overhead when disabled. Some of the typical providers are:

- *Function Boundary Tracing* provides an ability to insert probes at an entry to and a return from almost all kernel functions,
- *Statically Defined Tracing* utilises explicitly in-implementation specified probes with semantic meanings,
- *Syscall provider* makes available probes at an entry to and a return from each system call,
- *PID provider* allows to instrument arbitrary instructions in user-level processes.

The analysis code is written in the D language. It is a high-level language that resembles C and AWK. The code is compiled into a small reduced instruction set called D Intermediate Format (DIF). The kernel contains a DIF virtual machine to run the compiled code.

Listing 2.3 shows an example instrumentation of a user-level program. The executed program is a C compiler. The script is specified directly on the command line. It defines one probe to be inserted at the entry of the `malloc(3C)` function (`pid$target:libc:malloc:entry`). When the probe fires, the aggregating function `quantize()` is called. Using this function, the script measures a distribution of memory allocation sizes. In this case, the output reveals that the C compiler mostly performed many small allocations.

Listing 2.3: Using DTrace to obtain a distribution of memory allocation sizes

```
root@sol:~# dtrace -n 'pid$target:libc:malloc:entry
    { @ = quantize(arg0) }' -c 'cc hello.c'
dtrace: description 'pid$target:libc:malloc:entry '
matched 1 probe
dtrace: pid 4322 has exited


    value  ------------- Distribution ------------- count
        0 |                                         0
        1 |                                         2
        2 |@@                                       9
        4 |@@@@@@                                   33
        8 |@@@@@@                                   34
       16 |@@@@@@@@@@                               53
       32 |@@@@@                                    26
       64 |@@@@@@@                                  41
      128 |@                                        6
      256 |@                                        6
      512 |@                                        7
     1024 |                                         1
     2048 |@                                        6
     4096 |                                         0
     8192 |                                         0
    16384 |                                         2
    32768 |                                         1
    65536 |                                         0
```

2.3.3.3 Oracle Solaris Studio Analysers

Oracle Solaris Studio is a proprietary compiler suite and a software development product for Solaris (x86, AMD64, SPARC) and Linux (x86, AMD64), which is available free of charge from its website [45]. It is shipped with several dynamic program analysis tools, including performance, code and thread analysers. (Note that not all these tools are available in the Linux version.)

The performance analyser observes runtime characteristics of programs which helps programmers to find hotspots, bottlenecks and areas of high resource consumption in their product. The code analyser is a tool that incorporates both static and dynamic analysis techniques. The dynamic variant uses the static binary instrumentation. The analyser can detect memory leaks, out-of-bounds memory accesses, use of uninitialised memory and other similar errors. The thread analyser can find data races and deadlocks in programs. All analysers come with fancy GUIs.

Listing 2.4 illustrates how the code analyser can be used. A static analysis report is created by specifying the `-xanalyze=code` compiler option. A dynamic analysis report is created by running the actual program which is prior to that instrumented by the `discover(1)` tool. The reports are then viewed in a GUI tool.

Listing 2.4: Oracle Solaris Studio Code Analyser detecting a memory leak and a few other problems

```
setup@sol:~$ cat lintme.c
#include <stdlib.h>
int main(void)
{
  char *i = malloc(10);
  i[12] = 'a';
  return 0;
}
setup@sol:~$ cc -V
cc: Sun C 5.12 SunOS_i386 2011/11/16
setup@sol:~$ cc -g -xanalyze=code -o lintme lintme.c
setup@sol:~$ discover -a lintme
setup@sol:~$ ./lintme
setup@sol:~$ code-analyzer lintme
```

```
Static analysis
   ABW Beyond Array Bounds Write: i[12] @ line 5
   Memory Leak: 10 @ line 4
   MRC Missing malloc Return Value Check: malloc(10) @ line 4
Dynamic analysis
   ABW Beyond Array Bounds Write: at address 809001c (1 bytes) on the heap @ line 5
   Memory Leak: 10 bytes @ line 4
```

2.3.3.4 Pin

Pin is a framework for the dynamic binary instrumentation of programs [14, 48]. It is available for Linux and Windows and it supports the x86, AMD64 and Itanium architectures. It is a proprietary software developed and supported by Intel. It is free for a non-commercial use. Instrumentation tools (called Pintools) are written in C/C++ using Pin's rich set of APIs. Pin's goals are to provide an easy-to-use, portable, transparent and efficient instrumentation.

The power of Pin is fully demonstrated by the Intel Parallel Studio suite that uses Pin as an underlying engine [46, 50]. Tools included in this suite are able to find these defects:

- use of uninitialised values,
- use of invalid memory references,
- a mismatched memory allocation and deallocation,
- memory leaks,
- invalid use of stack memory,

- data races and deadlocks,
- performance bottlenecks.

Pin supports two instrumentation modes: a JIT-based one and a probe-based one. The JIT-based mode allows to completely instrument an application. The JIT compiler translates code from one ISA directly into the same ISA without using an intermediate representation. This technique is called the copy-and-annotate instrumentation (see Section 2.3.2.3). The compiled code is stored in a code cache and executed from there. The original code is never executed. Several methods are used to optimise the jitted code, including trace linking, register reallocation, inlining and liveness analysis. Using these methods, Pin achieves a very small slowdown, for example, the maximum slowdown for a basic-block counting is three times.

The probe-based mode provides an ability to instrument an application at the function-level only. It allows to wrap or replace functions in a program. It has almost none overhead because no code has to be jitted.

Pin also fully supports multi-threading (without any introduced serialisation) and similarly to debuggers, Pin can attach to an already running application.

Listing 2.5 shows a Pintool that logs information about all memory reads in a program. The function `main()` initialises Pin, opens a log file, registers the instrumentation function `Instr()` (called for each instruction before its translation) and the finalisation function `Fini()` and finally starts the observed program. The function `Instr()` inserts a call to the analysis function `MemoryRead()` if a given instruction reads memory. The function `MemoryRead()` simply logs a memory access. Finally, the function `Fini()` closes the log file after the program finishes.

Listing 2.5: Pintool for tracing memory reads

```
#include <stdio.h>
#include "pin.H"

static FILE *fo;

VOID MemoryRead(VOID *ip, VOID *addr, UINT32 size)
{
  fprintf(fo, "%p: %p %d\n", ip, addr, size);
}

VOID Instr(INS ins, VOID *v)
{
  if (INS_IsMemoryRead(ins))
    INS_InsertPredicatedCall(ins, IPOINT_BEFORE,
      AFUNPTR(MemoryRead), IARG_INST_PTR,
      IARG_MEMORYREAD_EA, IARG_MEMORYREAD_SIZE,
      IARG_END);
}

VOID Fini(INT32 code, VOID *v)
{
  fclose(trace);
}

int main(int argc, char **argv)
{
  PIN_Init(argc, argv);
  fo = fopen("trace.txt", "w");
  INS_AddInstrumentFunction(Instr, 0);
  PIN_AddFiniFunction(Fini, 0);
  PIN_StartProgram();
  return 0;
}
```

2.4 Comparison Between Static and Dynamic Analysis

A comparison of the two described analyses is not easy. In practice, a user should utilise both analyses (if it is possible) because each of them provides a different view on software.

Static analysis is based on more mathematical basis than its counterpart. It can be used to detect errors in an early stage of the development (before a program can be successfully executed), this is not possible for dynamic analysis. The static approach does not require any tests, whereas the dynamic method is effective only if there are available tests with good code coverage. Static analysis has problems to deal with non-deterministic inputs and usually has more false positives than the dynamic paradigm. Most modern dynamic analysis tools do not require any access to a program's source code, whereas the static analysis tools usually need such an access (unless they operate at the bytecode or machine code level). Also note that only the dynamic (runtime) binary analysis tools can deal with a dynamically generated code.

Chapter 3

Valgrind

3.1 Project Overview

Valgrind is a dynamic instrumentation framework and a set of associated tools which can detect many programming errors and also do profiling. Valgrind initially came into the public view in February 2002. It is licensed under the GNU GPL 2 [39]. It is currently available on the following platforms: Linux/x86, Linux/AMD64, Linux/ARM, Linux/PPC32, Linux/PPC64, Linux/S390X, Android/ARM, Darwin/x86 and Darwin/AMD64. There also exists an unofficial port to FreeBSD/x86 and FreeBSD/AMD64 [56].[2]

The original author of the project is Julian Seward, which received in 2006 a Google-O'Reilly Open Source Award for his work on Valgrind. The project itself won in 2004 a merit (bronze) Open Source Award and in 2008 a TrollTech's inaugural Qt Open Source Development Award for the best open source development tool.

Valgrind is a stable and mature system which is used worldwide by many programmers, ranging from beginners writing their course assignments to professionals landing rovers on Mars.

Note: this chapter is based on these main sources:

- the official documentation and other information available directly on the Valgrind website [55],
- Nethercote and Seward's research paper that describes how Valgrind works [17],
- Nethercote's Ph.D. dissertation about the dynamic binary analysis [16],
- the Valgrind source code.

[2] Several more attempts to port Valgrind to another platforms were done, including a port to NetBSD/x86, Solaris 8/x86 and even to Windows/x86. However, these ports were never finished. In the case of the Solaris 8 port, there is no public code available.

3.2 Available Tools

There are eleven tools available as a part of the official distribution of Valgrind:

- *Memcheck* is a memory error detector that catches uses of uninitialised memory, accesses to already freed memory, accesses off the end of allocated blocks and memory leaks.
- *Cachegrind* is a cache and branch prediction profiler. It simulates how a program utilises system caches and calculates a number of mispredicted branches.
- *Callgrind* is a call-graph generating cache and branch prediction profiler. It was originally an extension of Cachegrind and so some functionality overlaps with this tool.
- *Helgrind* and *DRD* are thread error detectors that catch misuse of the POSIX threads API, potential deadlocks and data races. Each of the tools uses a different analysis technique.
- *Massif* is a heap profiler that measures how much memory is used by client programs.
- *DHAT* is also a heap profiler but it does different profiling. It gathers statistics of lifetime and utilisation of memory blocks and helps to understand memory usage.
- *SGCheck* is a tool that can detect overruns of stack and global arrays. It is a complementary tool to Memcheck (their capabilities do not overlap).
- *BBV* is a tool that generates basic block vectors for use with the SimPoint analysis tool.
- *Lackey* is an example tool that illustrates how to do some simple program measurement and tracing.
- *Nulgrind* is the simplest Valgrind tool that performs no instrumentation or analysis. It is used by Valgrind developers for debugging and regression testing.

Note: DHAT, SGCheck and BBV are experimental tools.

There are also available several external Valgrind tools, for example, ThreadSanitizer – a race condition detector [21, 54].

Listing 3.1 shows an example of Memcheck's output. All messages from Valgrind are prefixed with a process identification number. Each error report contains a short description of the problem and a stack trace where the problem has happened.

Listing 3.1: Memcheck detecting a memory leak and an out-of-bounds access

```
setup@sol:~$ cat lintme.c
#include <stdlib.h>
int main(void)
{
  char *i = malloc(10);
  i[12] = 'a';
  return 0;
}
setup@sol:~$ cc -g -o lintme lintme.c
setup@sol:~$ valgrind --quiet --leak-check=full ./lintme
==632== Invalid write of size 1
==632==    at 0x8050CAC: main (lintme.c:5)
==632==  Address 0x156034 is 2 bytes after a block of size
10 alloc'd
==632==    at 0xFEFC0B84: malloc (vg_replace_malloc.c:271)
==632==    by 0x8050CA0: main (lintme.c:4)
==632==
==632== 10 bytes in 1 blocks are definitely lost in loss
record 1 of 1
==632==    at 0xFEFC0B84: malloc (vg_replace_malloc.c:271)
==632==    by 0x8050CA0: main (lintme.c:4)
==632==
```

3.3 Architecture and Implementation

This section describes Valgrind internals. It first provides a complete image how Valgrind works and continues with a description of individual subsystems. The focus has been put especially on subsystems that are important for the port. Parts not directly important for the port has been omitted or only superficially described.

3.3.1 Conceptual Overview

Valgrind consists of three parts: the Valgrind core (called Coregrind), tool plugins and the VEX library.

Coregrind forms a basic part of Valgrind. It contains an address space manager, a loader of client programs, a thread scheduler and a code dispatcher, a minimal libc implementation,[3] an internal memory allocator, a signal processing code, a debug information reader, a core dump creating code, a gdbserver and syscall wrappers. Many of these subsystems include parts which depend on an underlying operating system,

[3]Valgrind does not use the system C library to avoid any potential problems with having two copies of this library in the address space (one for Valgrind and one for a client program) or with sharing one library image.

especially the syscall wrappers are unique to each supported system. Coregrind is 150,000 lines of code.

The tool plugins do instrumentation of a client code and perform analysis of a program execution. Each tool is usually about 10,000 lines of code, Memcheck being the largest one with almost 20,000 lines.

The VEX library is responsible for translating a machine code into a processor-neutral intermediate representation and back to the machine code. This means that Valgrind uses the disassemble-and-resynthesise instrumentation (see Section 2.3.2.4). The library is 150,000 lines of code.

Valgrind is started by a simple launcher that determines which tool should be loaded. Each tool is a statically-linked executable compiled from Coregrind, a tool plugin and the VEX library. After the tool has been started, control is in Coregrind which first parses options and initialises several core subsystems, including the address space manager and the internal memory allocator. It then loads a client binary in the memory and initialises the tool plugin. Finally, more core subsystems are initialised, including the thread scheduler and the signal processing machinery. At this point, the translation of the first client instruction can begin. Note that Valgrind does not provide an ability for connecting to and instrumenting an already running program.

3.3.2 Translation

The translation covers all client code, including shared libraries and dynamically generated code. No original code is executed. The translation is always done per superblocks, in a just-in-time, execution-driven fashion. Each superblock is a portion of code that holds the single-entry, multiple-exit property.

The translation consists of several phases where most phases do some transformation of Valgrind's intermediate representation (IR). This IR is architecture-neutral, uses the single static assignment form, and is RISC-like (every instruction does only a primitive operation and it is the load/store type). However, the IR is not very RISC-like with regard to a number of instructions. To support all integer, FP and SIMD operations, it has to contain more than 200 primitive arithmetic and logical operations. Several special instructions, such as `cpuid` on x86, are also handled with a call to a C function that emulates the instruction.

Two similar IRs are used during the translation: a tree IR and a flat IR. The tree IR consists of statements representing actions with side-effects, such as assignments to temporaries or memory stores. The statements can contain arbitrarily complicated expressions which do not have any side effects, they merely represent complex arithmetics. The flat IR flattens these trees by saving computed values of expressions to temporary variables.

The following paragraphs describe all eight translation phases. The ones marked with an asterisk are architecture-specific. (The complete translation process can be nicely observed by invoking Valgrind with the `--trace-flags` option.)

- *Phase 1. Disassembly*: machine code → tree IR.* In the initial phase, an architecture-specific machine code is converted into the architecture-neutral IR. Each instruction is disassembled into one or more statements of the tree IR.
- *Phase 2. Optimisation 1: tree IR → flat IR.* This phase flattens the IR and performs several optimisations: redundant get and put elimination, copy and constant propagation, constant folding, dead-code removal, common subexpression elimination and even simple loop unrolling for intra-block loops.
- *Phase 3. Instrumentation: flat IR → flat IR.* This is the only tool-specific phase in the translation process. It is used by a tool for inserting instrumentation changes into the code.
- *Phase 4. Optimisation 2: flat IR → flat IR.* This is the second optimisation phase which is much simpler than the first one, only constant folding and dead-code removal is performed. The purpose of this phase is to optimise a potentially ineffective instrumentation code.
- *Phase 5. Tree building: flat IR → tree IR.* In this phase, the flat IR is converted back to the tree IR in preparation for instruction selection.
- *Phase 6. Instruction selection*: tree IR → instruction list.* This phase finally converts the IR into a list of instructions. The instructions do not reference to host registers but to virtual ones. The instruction selector uses a simple, greedy, top-down tree-matching algorithm.
- *Phase 7. Register allocation: instruction list → instruction list.* The virtual registers are replaced with host registers in a platform-independent manner. The allocator uses the linear scan algorithm [20, 24].
- *Phase 8. Assembly*: instruction list → machine code.* The final phase simply encodes and writes the selected instructions to a block of memory.

Listing 3.2 illustrates how one instruction is disassembled into the tree IR and then converted to the flat IR. In this case, it is a move instruction that can be interpreted in a C-like notation as `%eax = *(int*)(%eax + %esi * 1 + 8)`. The description of the IRs is as follows: The variables `t`*`num`* are temporaries. `IMark` is a no-operation that indicate where the instruction started, its address and length in bytes. `Add32` is a 32-bit add. `Shl32` is a 32-bit left-shift. `LDle` is a little-endian load. `GET` pulls a value out of a thread-state storage. `PUT` saves a value to the thread-state. The offsets 0 and 24 point to the registers EAX and ESI, respectively.

Listing 3.2: Disassembly to Valgrind's intermediate representation

Original instruction:
```
  0x10FB55:  movl 8(%eax,%esi,1),%eax
```

Disassembly into the tree IR:
```
  ------ IMark(0x10FB55, 4, 0) ------
  t0 = Add32(Add32(GET:I32(0),Shl32(GET:I32(24),0x0:I8)),
             0x8:I32)
  PUT(0) = LDle:I32(t0)
```

Optimisation 1, the tree IR is converted to the flat IR:
```
  ------ IMark(0x10FB55, 4, 0) ------
  t15 = GET:I32(24)
  t16 = GET:I32(0)
  t13 = Add32(t16,t15)
  t12 = Add32(t13,0x8:I32)
  t17 = LDle:I32(t12)
```

The translated code is stored in a translation table so it can be reused later without going through the complete translation process again. The translation table is a fixed-size, linear-probe hash table that uses the FIFO policy to select which code block should be evicted when the table becomes too full.

3.3.3 Executing Translations

The execution of the translated code is controlled by a dispatcher and a scheduler.

The dispatcher is a short assembly code (about 10 instructions on each platform) that is entered after the execution of one translated block has finished. The dispatcher uses a direct-mapped cache (of 32,768 entries) to find a following translated block for the execution according to the program counter value. If the search does not succeed then control falls in the scheduler.

The scheduler is a regular C code. When it receives control from the dispatcher it searches the complete translation table to find the required translated block. If the search succeeds then the dispatcher cache is updated accordingly and control is given back to the dispatcher. If the search is not successful then a new translation has to be prepared and then stored in the translation table and the dispatcher cache.

There are also several other cases when control is reclaimed by the scheduler, for example, when a maximum number of executed blocks has been reached so the scheduler can schedule another thread to run (see Section 3.3.4), or if a syscall instruction is encountered so a syscall wrapping code can be executed (see Section 3.3.6).

3.3.4 Threads

Valgrind supports multi-threaded programs. A client program running inside Valgrind has exactly the same process structure as it would have when executed natively (there are no extra threads). However, Valgrind allows only one thread to run at once. The scheduler, that runs in each thread, has to acquire a big lock before a thread can run any client code or manipulate any shared state.

The big lock is implemented as a self-pipe mechanism: the lock is released by writing a character into the pipe and acquired by the first thread reading the character from the pipe. Since these characters are read using the `read(2)` syscall, it is the kernel that decides which thread succeeds in the lock acquisition. This means that even though Valgrind serialises threads, the kernel is still in charge of scheduling them.

A thread, holding the big lock, executes at maximum 100,000 translated basic blocks before it gives control back to the scheduler which releases the lock and waits again on the lock acquisition. A problem arises if the thread needs to perform a possibly blocking syscall. In such a case, the thread releases the big lock and makes the syscall. After the syscall finishes, the thread has to reacquire the lock again before it can continue the execution.

3.3.5 Signals

Handling Unix signals is generally not easy [38]. In the case of Valgrind, it is especially complicated.

When a client program registers a signal handler, the kernel receives an address pointing to a client code that should be called to handle the signal. This could cause that the client code is executed natively. To prevent this behaviour, Valgrind intercepts all signal-related syscalls and supplies to the kernel its own signal handlers that appropriately wrap client handlers.

The next problem is that an analysis code and an original code are closely interspersed and cannot be arbitrary separated by a signal delivery. For example, Memcheck tracks definedness of every bit of a client data. If a value of a client bit is set then an associated definedness bit has to be updated. However, it is not possible to perform both operations in one atomic instruction, two are usually required. If a signal is delivered between these two instructions then a signal handler is executed while Valgrind is in an inconsistent state. Therefore Valgrind blocks signals while executing a translated code and polls for them between code block executions.

However, some signals cannot be postponed in this way. Synchronous signals that are instruction-generated, for example, `SIGILL` or `SIGSEGV` have to be processed immediately when they occur. Thus these signals are not blocked and their handler does a longjump out of the translated code to the scheduler which proceeds them appropriately.

Finally, signals cannot be blocked while a thread is in a blocking syscall. For example, if `SIGINT` was blocked while the thread is stuck in the `read(2)` syscall then the program cannot be terminated with the `Ctrl-C` combination. (This example assumes that the program is set to terminate on `SIGINT` and that `Ctrl-C` sends `SIGINT` to the process.) Valgrind contains a non-trivial assembly code to unblock/reblock the signals before/after making a blocking syscall and an associated C code to deal with signals received while executing this code.

3.3.6 System Calls

Valgrind is a user space tool which means that it cannot track a program into the kernel. When a syscall happens, Valgrind has to be explicitly informed about effects of this syscall. This includes which registers and memory is read and written by the syscall and if there are any changes to open file descriptors or memory mappings. All these effects are described in a syscall wrapper machinery. Even though writing a single syscall wrapper is relatively easy, there are several hundreds of different syscalls that are available on each platform. Therefore this machinery makes a huge part of Coregrind, for example, the syscall wrappers for Linux/x86 consist of around 10,000 lines of code.

Listing 3.3 illustrates a simple syscall wrapper, namely the `time(2)` wrapper. The function `PRE(sys_time)()` is called before the syscall is performed. In this case, it contains three macros: `PRINT()` outputs information about the syscall if the `--trace-syscalls` option is used, `PRE_REG_READ1()` informs Valgrind about parameters that are read by the syscall and `PRE_MEM_WRITE()` is used to mark a block of client memory that is about to be written by the syscall. The function `POST(sys_time)()` is called after the syscall finishes. The macro `POST_MEM_WRITE()` marks a block of client memory initialised by the syscall.

Listing 3.3: Syscall wrapper for `time(2)`

```
PRE(sys_time)
{
   /* time_t time(time_t *t); */
   PRINT("sys_time ( %#lx )", ARG1);
   PRE_REG_READ1(long, "time", time_t *, t);
   if (ARG1 != 0) {
      PRE_MEM_WRITE("time(t)", ARG1, sizeof(vki_time_t));
   }
}

POST(sys_time)
{
   if (ARG1 != 0) {
      POST_MEM_WRITE(ARG1, sizeof(vki_time_t));
   }
}
```

3.3.7 Other Facilities

3.3.7.1 Debug Information Reader

Valgrind relies on debug information to provide accurate error reports to its users. Stripped programs and libraries limit usefulness of provided reports. Valgrind is able to read debug information stored in several formats, including STABS [23, 52] and DWARF [33], which are the two most common formats used on Unix-like systems, and PDB [31], which is a format used on Windows systems. The PDB support allows Valgrind to work with Windows applications running with Wine.[4]

3.3.7.2 Function Replacement and Function Wrapping

Valgrind contains an advanced facility to redirect or wrap functions. The difference between the two is that wrapping allows to call an original function from a replacement function. This facility is primarily used by Valgrind itself. For example, Memcheck redirects malloc-family functions to do a leak detection. It is possible to intercept any function that has a known entry address.

3.3.7.3 Client Requests

Client requests allow manipulating and querying a state of Valgrind from a client program. A user includes `valgrind.h` into her program and uses macros available in this file. A request is encoded using a special preamble which is recognised by Valgrind in the translation process. The preamble is a series of instructions that is unlikely to appear in a normal program. For example, the x86 preamble is `roll $3, %edi; roll $13, %edi; roll $29, %edi; roll $19, %edi`.[5] A program containing the client requests can still be run normally without Valgrind.

The requests allow a program to determine whether it is running under Valgrind, discard translations cached by Valgrind (useful for runtime code generation), call functions natively, query a number of detected errors, inform Valgrind about memory pools and stacks (to give Valgrind more information about their usage in a program), or print a stack backtrace.

3.3.7.4 Core Dumps and Gdbserver

Valgrind re-translates a client program and hides its execution under a big volume of other code which causes that an attached debugger sees the internals of Valgrind instead of the client ones. Valgrind provides two facilities to improve this situation.

First, if a crash occurs in a client code then Valgrind creates a core dump that corresponds to an actual client state and that contains only memory segments of the

[4]Wine is a compatibility layer that reimplements the Windows API and allows to run Windows programs on Unix-like systems [59].

[5]Instruction `rol b, r` rotates `r` to the left by `b` bits. In this case, rotating a content of the EDI register by 64 bits ($3 + 13 + 29 + 19 = 64$) can be seen as a fancy no-operation instruction.

client (Valgrind-allocated segments are omitted). A regular debugger can then be used for post-mortem analysis on such a core dump.

Second, since version 3.7.0 (released on November 2011) Valgrind implements a gdbserver that allows connecting to a running Valgrind instance using the GNU Project Debugger (GDB) [37]. The gdbserver exports information about a client program and communicates with GDB using the GDB Remote Serial Protocol [36]. This facility also provides a way to interactively query the Valgrind core or tools, for example, to do an incremental leak search under Memcheck.

3.4 Limitations

Valgrind has several limitations and problems that arise from its design and implementation.

The main limitation is that a program under Valgrind runs 10-50 times slower than natively. It is so because of the complicated translation process and heavyweight instrumentation done by some tools. This slowdown prohibits the use of Valgrind on production systems and may eventually hide some bugs in a client program. Thread serialisation done by Valgrind poses another performance loss. Luckily, some attempts to make Valgrind multi-threaded have recently begun.

The main problem of analysis performed by the tools are false positives. For example, Memcheck can be tricked by a highly optimised code that contains bit tricks or partial loads. Listing 3.4 shows a Memcheck false positive. It can be easily proved that the tool is incorrect in this case because the variable `zoo` is initialised to zero at the `printf(3C)` statement. Because of this, Valgrind developers advise users to use only low optimisation levels if they are compiling a program to be run under Valgrind.

Listing 3.4: Example of a Memcheck false positive

```
setup@sol:~$ cat fool.c
#include <stdio.h>
int main(void)
{
  int zoo;
  int i = zoo;
  zoo = zoo ^ i;
  printf("zoo=%d\n", zoo);
  return 0;
}
setup@sol:~$ cc -g fool.c -o fool
setup@sol:~$ objdump -d fool
[snip]
08050c8c <main>:
 8050c8c:   push   %ebp
 8050c8d:   mov    %esp,%ebp
 8050c8f:   and    $0xfffffff0,%esp
 8050c92:   sub    $0x20,%esp
 8050c95:   mov    0x1c(%esp),%eax       # zoo -> %eax
 8050c99:   mov    %eax,0x18(%esp)       # %eax -> i
 8050c9d:   mov    0x18(%esp),%eax       # i -> %eax
 8050ca1:   xor    %eax,0x1c(%esp)       # %eax ^ zoo -> zoo
 8050ca5:   mov    0x1c(%esp),%eax
 8050ca9:   mov    %eax,0x4(%esp)
 8050cad:   movl   $0x8050d3c,(%esp)
 8050cb4:   call   8050a54 <printf@plt>
 8050cb9:   mov    $0x0,%eax
 8050cbe:   leave
 8050cbf:   ret
[snip]
setup@sol:~$ valgrind --quiet --track-origins=yes ./fool
[snip]
==791== Syscall param write(buf) points to uninitialised byte
==791==    at 0xFEF54965: __write (in /lib/libc.so.1)
==791==    by 0xFEF20C82: _xflsbuf (in /lib/libc.so.1)
==791==    by 0xFEF131CC: _ndoprnt (in /lib/libc.so.1)
==791==    by 0xFEF14366: printf (in /lib/libc.so.1)
==791==    by 0x8050CB8: main (fool.c:7)
==791==  Uninitialised value was created by a stack allocation
==791==    at 0x8050C92: main (fool.c:3)
==791==
zoo=0
```

Chapter 4

Port Implementation

This chapter describes how the port is implemented. It follows closely the layout of the previous chapter and covers what changes were necessary in each relevant part of Valgrind. The text also discusses some differences between Solaris and Linux. Knowing them is important in comprehending what needed to be done differently in this port compared to the Linux version.

Porting Valgrind to a new platform requires knowledge in many areas. In the Solaris/x86 case, it involves understanding internals of Valgrind, specific parts of the Solaris kernel, the runtime linker and the standard C library as well as understanding the ELF format and the x86 processor architecture. Knowledge of the STABS and DWARF formats is also partly necessary.

This chapter assumes that the reader is familiar with the x86 processor architecture [12].

The port is based on the Valgrind source code as of 6th April 2012 (Valgrind revision 12495, VEX revision 2274).

4.1 Build System

The Valgrind build process is based on the GNU Autotools (primarily on Autoconf [25] and Automake [26]) and the make utility. Minor modifications in this system were necessary. The changes include building only the x86 version even if the AMD64 system is present, or fixing linking options of some regression tests (for example, if a test uses socket related functions then the test has to be linked against the socket and nsl libraries).

The only notable addition to the build system is a Solaris specific linker script. Each linker script is a small Perl program whose main purpose is to tell an underlying linker to set an alternate load address of the tool executables. This is necessary because the tools are regular, position-dependent executables and if a tool was loaded at the default address (0x8000000 on Solaris) then it would clash with the most client programs (which are usually also position-dependent executables). There is no portable way to change the load address so there is a linker script for each supported operating system. The Solaris version of this script creates a mapfile containing appropriate linker directives to modify the default behaviour and passes this file to the Solaris linker. The mapfile format version 2 is used [18].

4.2 Standard Library

Valgrind implements a minimal subset of the standard C library (see Section 3.3.1). This includes a variety of functions, some of which are platform specific because they require assistance of the kernel. Most of these functions represent low-level functions from the POSIX standard [61], which are usually implemented directly by the kernel so it is enough to just make an appropriate syscall. There are a few exceptions, for example, there is no `open(2)` syscall on Solaris 11, therefore the `openat(2)` syscall with the first parameter set to `AT_FDCWD` is used instead. This port makes syscalls via the `int $0x91` instruction which is the most portable way how to enter the Solaris kernel on the x86 platform.

4.3 Client Program Loading

Loading a client program into the memory constitutes a significant part of the initialisation procedure which involves several steps and various modules.

A prerequisite of the loading process is the initialisation of the address space manager which is a component that maintains information about current (virtual) memory mappings. When this module is initialised it needs to obtain mappings that are already in place. This is done by a syscall (Darwin) or by accessing the `/proc` filesystem (Linux and Solaris). In the Solaris case, current mappings are read from the file `/proc/self/xmap` which is a binary file containing an array of `prxmap` structures. Each structure describes one contiguous address segment.

The Solaris port supports reading and mapping ELF[6] objects. The actual process is very similar to what the kernel does: a client binary is read from a disk, program headers are parsed, load segments are mapped into the memory, if an interpret is required then it is loaded too, and finally, a stack is initialised. Figure 4.1 shows how the virtual address space is partitioned for a typical program. It contains some differences compared to a normal run. The runtime linker is loaded in the lower part of the address space, while it is normally mapped at the opposite end (this is not particularly important as the runtime linker is a position-independent code). Valgrind is loaded at the address 0x38000000, this is determined by the linker script (see Section 4.1). The client stack, which is allocated by Valgrind, is positioned below the Valgrind image. The initial stack set by the kernel is left unused, Valgrind switches to its own statically allocated stack.

[6]Executable and Linkable Format (ELF) is a common standard file format for code objects on Unix-like operating systems, including Solaris [53].

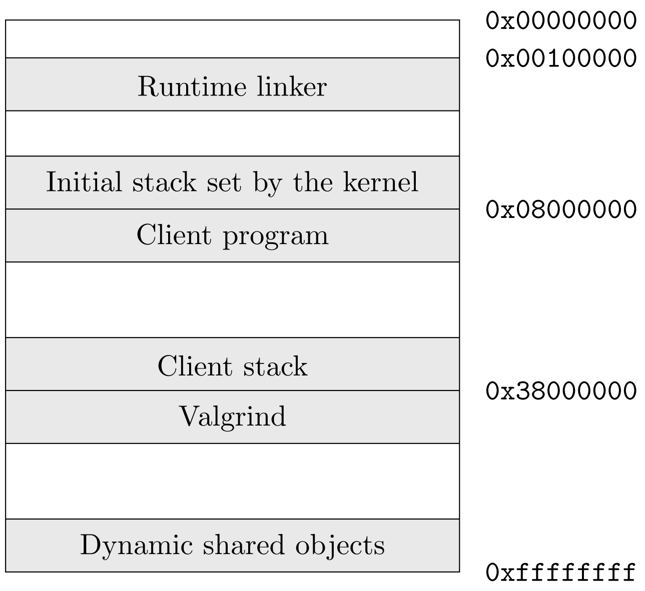

Figure 4.1: Virtual address space partitioning

The ELF loader has been extended to understand the `PT_SUNWDTRACE` program header. This header contains an address of a scratch area that is used by the DTrace fasttrap provider. The address represents an initial value of the thread pointer (see Section 4.5).

The final stage of the loading process initialises the client stack. Figure 4.2 displays a content of such a stack. It contains a number of arguments to the program (argc), string pointers to actual arguments (argv), string pointers to environment variables (envp), an auxiliary vector (auxv) and a string table.

All values are usually derived from values that the kernel passed to Valgrind on its initial stack. However on Solaris, the kernel does not create any auxiliary vector if an executed program is a statically-linked program, which is the Valgrind case. Therefore the auxiliary vector has to be built from scratch. The vector stores information that is primarily intended to be used by the runtime linker. It contains a platform identifier, a name of an executed program, an address referring to a memory location where program headers of an executed program are mapped, an address where an interpreter is mapped, a size of memory pages, hardware capabilities and potentially other similar values. Most of the values are either constants or originate from the ELF loader. The hardware capabilities are currently obtained by using a private kernel interface (which is originally provided for the `isainfo(1)` command). This should be eventually changed so that the hardware capabilities reflect the properties of the virtual CPU that Valgrind synthesises.

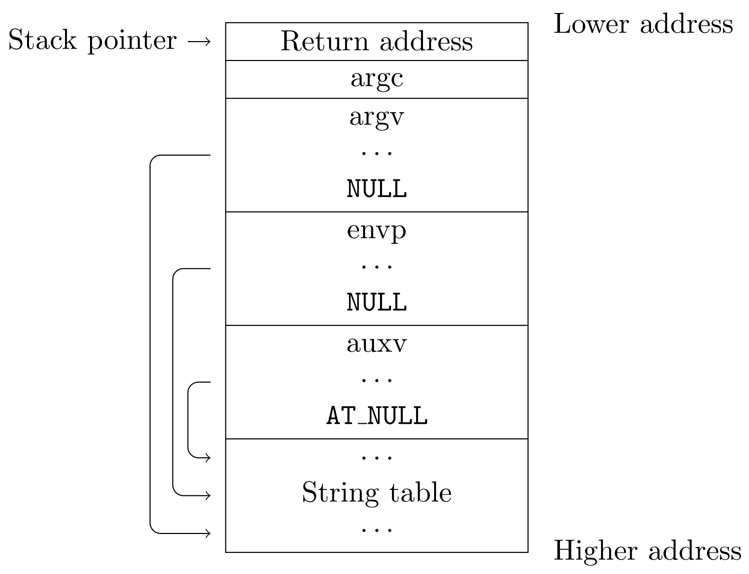

Figure 4.2: Initial stack layout

4.4 Executing Translations

Valgrind contains a special dispatcher for each supported platform. The Solaris/x86 dispatcher is an exact copy of the Linux/x86 version. The negligible change is that it does not create the `.note.GNU-stack` section which in the Linux case notifies a linker that this object does not require an executable stack.

4.5 Threads

The multi-thread support is closely tied to two issues: thread-local storage implementation and a logic associated with creating new threads.

On both Solaris/x86 and Linux/x86, the thread-local storage implementation utilises segmentation. The FS and GS registers each hold a segment selector which points to a segment descriptor in the GDT.[7] The segment base address field in the segment descriptor points to the thread-local data. This address is called a thread pointer. The GS register is used by the standard C library and the FS register is utilised by emulators or similar programs, for example, by Wine. A user process cannot directly access the GDT, a kernel interface has to be used. Therefore Valgrind

[7]Global Descriptor Table (GDT) is an array of segment descriptors that define aspects of various memory areas used during a program execution. (Note: Besides segment descriptors, the GDT can also hold other structures, for example, Task State Segment.)

simulates the GDT because some tools, for example, Memcheck, depend on knowing the linear address of every memory access.

Both the Linux and Solaris kernels maintain an individual GDT per each CPU. The kernels swap only a few segment descriptors when a new thread is about to run. On Solaris, two of these loaded descriptors are important for the thread-local storage implementation, the `GDT_LWPFS` and `GDT_LWPGS` descriptors. Programs can use the `lwp_private(2)` syscall to define the base address in these descriptors which also causes that the `LWPFS_SEL` or `LWPGS_SEL` segment selector (depending on which descriptor is being set) is stored in the FS or GS register, respectively.

The Solaris port follows an approach of the Linux version when it simulates an individual GDT for each thread. This method has relatively large overhead (64 kB per thread) but it does not require any modification to the generic Valgrind code (which is required when a shared GDT is used) and it will work even when the Valgrind big lock limitation is lifted.

New threads are created on Solaris using the `lwp_create(2)` syscall. The first parameter of this syscall specifies a complete execution context of a new thread. The context holds values of all processor registers and flags. Valgrind cannot pass this context directly to the kernel because that would lead to the execution of an original client code. The `lwp_create()` syscall wrapper contains a code to create a new Valgrind thread-state, allocate a stack that will be used by Valgrind to run this thread, and save values from the context to the VEX guest state (part of the thread-state). This last part is problematic because there are minor differences between the real processor context and the VEX guest state (see Section 4.7.1.2). Then the `lwp_create()` wrapper fills a new context that sets the processor to run a Valgrind thread wrapper and finally makes the `lwp_create()` syscall with this own supplied context.

In comparison, new threads on Linux are created using the `sys_clone(2)` syscall which has a similar behaviour to `fork(2)`. It duplicates a calling thread, both threads continue at a point of the call, but a return value to the parent is a pid of the child thread while a return value to the child is zero.

4.6 Signals

Signal implementation differs in a few aspects across various Unix-like operating systems. The most significant difference (from the Valgrind point of view) is how an original execution context is restored after a signal handler finishes.

On Linux/x86, the kernel saves into the return-address slot on a signal frame an address of a restorer function. When the execution of the signal handler finishes, the return address is popped of the stack and control is transferred to the restorer which makes the `sigreturn(2)` syscall to restore an original context. The restorer code comes from the kernel and originally used to be saved directly on the signal stack, but because that prevented the stack from being non-executable, the restorer is now located in so called Virtual Dynamically-linked Shared Object. It is a special

ELF object that is mapped into a process's address space by the kernel. It is also possible to set a custom restorer function when a signal handler is registered. The Linux/AMD64 implementation is similar but the standard C library is required to always set the custom restorer function.

On Solaris, if a program wants to return from a signal handler to an original context then it has to do so explicitly using the `setcontext(2)` variant of the `context()` syscall. Typically, programs use the standard C library which wraps user signal handlers and the wrapper makes the neccessary `setcontext()` call.

The discussed difference is important in two contexts. First, it leads to a small change in the Valgrind signal handler, an explicit `setcontext()` call needs to be added at the end of the handler. Second, it affects how Valgrind delivers signals to client programs.

Valgrind synthesises the signal delivery to client programs. This means that Valgrind builds a complete signal frame and takes special care about returning from a signal handler. Figure 4.3 shows a native signal frame on Solaris. It contains an invalid return address, a signal number, an old execution context and optionally extended information about a signal. The Solaris port follows exactly this layout, no extra data is saved on the stack. The context structure is filled by values from the VEX guest state. When a client program returns from a signal handler, it makes the `setcontext()` call and Valgrind restores the original context in the `PRE` wrapper of this call.

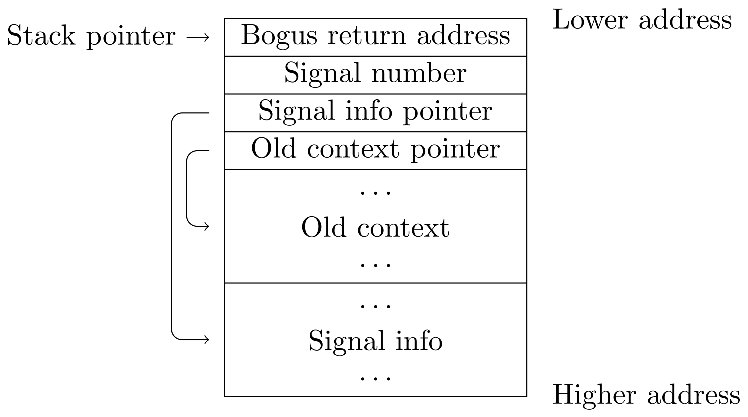

Figure 4.3: Signal frame

In comparison, the Linux version of Valgrind builds a less accurate stack frame. It does not store the FP and SSE values in the context structure and it saves extra Valgrind data in the frame which are used (instead of the context structure) to restore an interrupted execution point. Valgrind utilises a custom restorer which makes the `rt_sigreturn(2)` syscall. The `PRE` wrapper of this syscall then restores the original context.

4.7 System Calls

There are many ways how a user space program can make a syscall on Solaris/x86. It can be made via a call gate, several interrupt gates, Intel's `sysenter` or AMD's `syscall` instructions. The port does not support all of them but only a reasonable subset. The call gate is not supported because it poses a legacy way how to enter the kernel and is never utilised by the Solaris standard C library. The 0x80 interrupt gate is also not supported because it is used merely to run Linux programs on Solaris. Finally, the `syscall` instruction cannot be used because the VEX library synthesises the Intel x86 processor.

Figure 4.4 visualises how a syscall is made from the point of view of a user space program. Before the syscall, up to eight parameters are stored on a stack and a syscall number is saved in the EAX register. If the syscall succeeds then the carry flag is cleared and a return value is to be found in the EDX and EAX registers. If the syscall fails, the carry flag is set and the EAX register contains an error number.

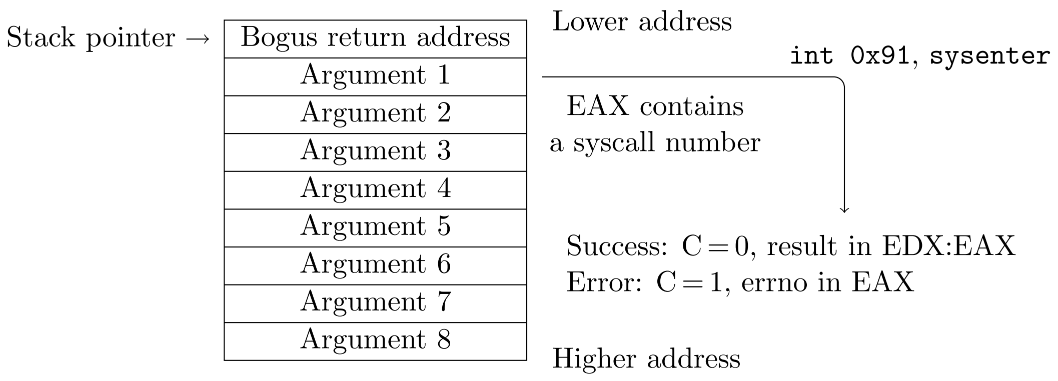

Figure 4.4: Syscall from the point of view of a user space program

Solaris also supports other type of syscalls which are called fasttraps and which can be used via the `0xd2` interrupt gate. These syscalls do not take any parameters (except for a syscall number in the EAX register) and never fail. This allows them to go through a fast path in the kernel. There are only a few fasttraps, for example, one is used by `clock_gettime(3C)`.

To support Solaris syscalls in Valgrind, several changes were necessary. The VEX library has been extended to disassemble the `int 0x91` and `int 0xd2` instructions. The syscall-wrapper main module has been modified to know where parameters of the Solaris syscalls are stored, where a result is saved and how to recognise if a syscall failed or not. An assembly code for making blocking syscalls has been added, it is based on the Darwin/x86 version with minor modifications. Finally, the syscall wrapper machinery for the Solaris syscalls has been implemented. Tables 4.1 and 4.2 summarise syscalls currently supported by this port (about 40 % of all syscalls are supported). The note column contains hints about a given syscall. The legacy remark means that a syscall is available only on Solaris 11 Express but not in the final Solaris 11. A section number then means that a syscall is somehow important and more information about the syscall can be found in that section.

Difficulty of writing syscall wrappers range rapidly. A trivial case is when Valgrind already provides a required wrapper. This happens when a syscall is common on all supported systems. If it is not the case then a new wrapper has to be written from scratch. A problem is that there is limited information on many syscalls. The documentation in the second section of manual pages describes an interface implemented by the standard C library which does not always match a real syscall interface. For quite a lot syscalls, there is no documentation at all. In such cases, the only way how to comprehend the syscall interface is by reading the source codes of the kernel and the standard C library.

4.7.1 Important System Call Wrappers

This section discusses a few examples of syscalls that are for some reason interesting for this port, for instance, a modification to the generic Valgrind code was necessary to handle them correctly. Information about the `lwp_create()` and `lwp_private()` syscalls was already presented in Section 4.5.

4.7.1.1 ioctl

The `ioctl(2)` syscall provides an interface to control devices. It takes three parameters: a file descriptor, a request identifier and an optional request-dependent value. Implementing a wrapper for this syscall is complicated because there are literally hundreds of various requests and Valgrind needs to know what memory is read or written by every request that is used by a client program. Otherwise Valgrind may hide some errors or report false positives.

Solaris provides the `sys/ioccom.h` header file that contains macros to encode information about the third parameter in the request identifier. This encoding can express that the third parameter is a pointer to a memory block, if the block is read or written and a size of the block. Unfortunately this feature has been rarely utilised by Solaris developers so it is not much use for this wrapper and most requests still have to be described manually.

4.7.1.2 context

The previous sections 4.5 and 4.6 already discussed where execution contexts are involved on Solaris. The `context()` syscall is used to save a current execution context (in the `ucontext_t` structure) and later restore it (`getcontext(2)`, `setcontext(2)`). The port completely simulates this syscall which means that it is processed directly by Valgrind and control is never passed to the kernel. It is necessary to do so because if the syscall was processed by the kernel then a saved context would contain a current Valgrind context instead of a client program.

The context structure consists of a signal mask, a stack declaration and a processor state. The problematic part is the processor state. Valgrind synthesises an x86 CPU that is very similar to a physical x86 CPU but a representation of the EFLAGS register, FPU flags and the ST(x) registers is different. In order to get a required format of these values, several functions to convert between the VEX representation and the real format were exported from the VEX library. These functions provide abilities to save/restore the EFLAGS register, the x87 state and the MXCSR register. However, there is a problem with the EFLAGS register. The VEX library does not simulate this register directly, instead it uses four 32-bit values that represent the last arithmetic operation and operands used. It is possible to calculate the EFLAGS value from these four values but it does not work vice versa without a loss of precision. Therefore this port saves these four values (plus a check sum of them) into unused slots in the context structure. It means that the semantics of this syscall is slightly changed when a program is run under Valgrind.

A further modification, that was required to handle this syscall, is adding two new events which the core sends to the tools. They inform the tools that a VEX guest state value is saved into a memory block or loaded from a memory block. Currently, the only consumer of these events is the Memcheck tool which requires them to track the value definedness. Appropriate handlers were added to Memcheck.

4.7.1.3 mmap

The `mmap(2)` syscall establishes a mapping between a process's address space and a memory object. The Solaris version of this syscall allows to make a mapping with a specified alignment. The kernel fulfils the alignment requirement but has a freedom to determine where the mapping should occur.

Valgrind intercepts `mmap()` calls and if it finds that a mapping does not have a fixed address then the address space manager assigns the address to the mapping. The kernel is then entered with a fixed mapping request. This is necessary because the kernel does not know about memory areas that Valgrind does not map but reserves them for other use, for example, a reservation exists for a memory area into which a grow-down client stack is expanded. The code of the address space manager has been extended to calculate the fixed address for this Solaris specific alignment extension.

4.7.1.4 mmapobj

The `mmapobj(2)` syscall is similar to `mmap(2)`. It establishes a set of mappings between a process's address space and a file. Optionally, it can also interpret the file and map it according to the rules of that file format. It is used mainly by the runtime linker to map dynamic libraries. It is worth mentioning that this syscall is not to be found on any other Unix-like system.

From the Valgrind point of view this syscall creates two problems which have to be dealt with in the `POST` wrapper. The syscall can make more than one mapping at once which is something that the sanity checks in the address space manager do not expect. To fix this issue the checks are temporarily disabled until the address space manager is transferred to a consistent state. The second problem is that Valgrind does not know which mappings have been created by the call which means that the `/proc/self/xmap` file has to be reread to update the address space manager state. (Note that the syscall actually returns which mappings have been made but not all mappings are always correctly recorded.)

4.7.1.5 schedctl

This undocumented syscall returns a pointer to a per-thread structure `scshared_t` that is used for a communication between the kernel and the standard C library. The structure is allocated in a memory page that is mapped by the kernel in the process's address space.

This interface is problematic for Valgrind, for example, it allows the standard C library to block all signals by setting the `sc_sigblock` member of the structure. The port cannot let this happen because it would completely mess up the Valgrind signal machinery. Therefore the port does not support this interface and returns the value zero when this syscall is made. This value tells the standard C library that the interface is not available, the library then uses a fallback code instead of using the interface.

4.8 Other Facilities

4.8.1 Debug Information Reader

The Solaris port supports reading the DWARF, STABS and PDB debug information formats. Only minor modifications to the reader were necessary.

The ELF reader has been extended to interpret the `.SUNW_ldynsym` section. This section contains a table of local function symbols and allows to get a useful stack trace for stripped programs that do not have the `.symtab` section [58].

4.8.2 Function Replacement

All function redirects installed by Coregrind and the tools are OS specific, but in fact they are very similar for all supported operating systems. Solaris is no exception. Replacements are registered for overly optimised string and memory functions located in the runtime linker and the standard C library. Memcheck's redirects for malloc-family functions are installed for the allocator available in the standard C library. Other memory allocators (bsdmalloc, libumem, mtmalloc, watchmalloc) are not supported. This follows the general Memcheck behaviour when only a main system allocator on each OS is supported.

4.8.3 Client Requests

The platform specific part of the client requests implementation is shared between the Darwin/x86, Linux/x86 and Solaris/x86 ports.

4.8.4 Core Dumps and Gdbserver

The core dump functionality is not currently implemented by this port. The gdb-server is also not supported because it relies on the `ptrace(2)` syscall. The Solaris kernel does not implement this syscall but instead it provides the `/proc` file system that can be used to simulate the `ptrace()` behaviour. The port does not yet contain a code to do so.

4.9 Tools

The tools are mostly platform independent, therefore not many changes were necessary in this area. The modifications to the Memcheck tool were already discussed in Sections 4.7.1.2 and 4.8.2. The only not mentioned change in this tool is that Memcheck on Solaris has to threat a `brk(2)` allocated memory as initialised (which is a correct behaviour because the brk-allocated memory should always be zeroed). Not doing so causes a huge number of false positives in several programs that utilise the `brk()` memory management.

Out of eleven tools, two thread error detectors (DRD and Helgrind) are currently not available. They are disabled because they report many errors in the standard C library (which were not sorted out yet) and do not support Solaris threads (the `thr_*()` functions). Therefore they are not much useful to end users yet.

Sysno	Name	Note	Sysno	Name	Note
1	`exit`		97	`sigaltstack`	
3	`read`		98	`sigaction`	
4	`write`		99	`sigpending`	
5	`open`	legacy	100	`context`	4.7.1.2
6	`close`		107	`waitid`	
10	`unlink`	legacy	112	`priocntlsys`	
12	`chdir`		113	`pathconf`	
13	`time`		115	`mmap`	4.7.1.3
17	`brk`		116	`mprotect`	
18	`stat`	legacy	117	`munmap`	
19	`lseek`		121	`readv`	
20	`getpid`		122	`writev`	
21	`mount`		127	`mmapobj`	4.7.1.4
22	`readlinkat`		128	`setrlimit`	
24	`getuid`		129	`getrlimit`	
27	`alarm`		131	`memcntl`	
28	`fstat`	legacy	137	`sysconfig`	
29	`pause`		139	`systeminfo`	
33	`access`		142	`forksys`	
37	`kill`		146	`yield`	
42	`pipe`		159	`lwp_create`	4.5
43	`times`		160	`lwp_exit`	
47	`getgid`		161	`lwp_suspend`	
50	`sysi86`		162	`lwp_continue`	
54	`ioctl`	4.7.1.1	163	`lwp_kill`	
59	`execve`		164	`lwp_self`	
60	`umask`		165	`lwp_sigmask`	
62	`fcntl`		166	`lwp_private`	4.5
66	`fstatat`		167	`lwp_wait`	
67	`fstatat64`		175	`llseek`	
68	`openat`		183	`pollsys`	
77	`lwp_park`		199	`nanosleep`	
80	`mkdir`	legacy	206	`schedctl`	4.7.1.5
90	`readlink`	legacy	209	`resolvepath`	
95	`sigprocmask`		213	`getdents64`	

Table 4.1: Supported system calls, part 1

Sysno	Name	Note
215	`stat64`	legacy
216	`lstat64`	legacy
217	`fstat64`	legacy
221	`getrlimit64`	
225	`open64`	legacy
229	`getcwd`	
230	`so_socket`	
231	`so_socketpair`	
232	`bind`	
233	`listen`	
234	`accept`	
235	`connect`	
238	`recvfrom`	
239	`recvmsg`	
240	`send`	
241	`sendmsg`	
244	`getsockname`	
245	`getsockopt`	
246	`setsockopt`	
254	`uucopy`	

Table 4.2: Supported system calls, part 2

Chapter 5

Evaluation

5.1 Testing

The testing of the port was done mainly by the Valgrind test suite. It consists of unit and regression tests which cover most of the functionality that Valgrind provides. Each test is a small program that is run under Valgrind and its output is compared to an expected output. The test suite also contains a few real-life applications, for example, the bzip2 data compressor.

Table 5.1 summarises results for different modules. The failing tests are caused by minor differences in expected outputs. A few tests also fail because they are non-portable and depend on a specific system behaviour or because they make a syscall that does not have yet implemented an associated syscall wrapper. The Helgrind, DRD and Gdbserver tests are disabled because their functionality is not yet supported.

Module	Tests	Fails
Coregrind+VEX	130	29
Memcheck	153	19
Cachegrind	6	0
Callgrind	13	0
Helgrind	unavailable	
DRD	unavailable	
Massif	34	0
DHAT	0	0
SGCheck	6	4
BBV	4	0
Lackey	1	0
Gdbserver	unavailable	

Table 5.1: Results of the Solaris port in the Valgrind test suite

5.2 Bug Example

Listing 5.1 shows a bug found by the Memcheck tool in the Solaris standard C library, specifically in the implementation of the `snprintf(3C)` function. This bug actually seems to be a known feature of the library where a hack is used to extend the `FILE` structure.

The extended structure is called `xFILE` and it consists of the `FILE` and `xFILEdata` members. The structure is not available outside the library. However, if there is a pointer to the original file structure, it can actually be a pointer to the extended variant. The library needs a way how to recognise which structure is hidden behind the pointer. A hack is used to do so. The first member of the `xFILEdata` structure holds a value that is calculated as XOR between an address of the structure and the value 0x63687367. When the library obtains a pointer to one of the file structures, it decides which structure is present according to this member.

The `snprintf()` function allocates on its stack the `FILE` structure and passes a pointer to this structure to other functions in the library. When the `getxfdat()` function tries to determine which structure is present, it does a conditional jump according to a value that is located past the allocated `FILE` structure. This value is in this case uninitialised and Valgrind reports the error. This hack does not normally cause any harm, but under specific conditions, it can.

This bug nicely demonstrates what problems is this port able to find on Solaris. Note that the 64-bit version of the library is not affected by this bug.

Listing 5.1: Bug found by Memcheck in the Solaris standard C library

```
setup@sol:~$ cat bug.c
#include <stdio.h>
int main(void)
{
  char buf[64];
  snprintf(buf, sizeof(buf), "Hello");
  return 0;
}
setup@sol:~$ cc -g bug.c -o bug
setup@sol:~$ valgrind --quiet --track-origins=yes ./bug
==857== Conditional jump or move depends on uninitialised value
==857==    at 0xFEF20AAF: getxfdat (in /lib/libc.so.1)
==857==    by 0xFEF20B47: _realbufend (in /lib/libc.so.1)
==857==    by 0xFEF0FB7A: _ndoprnt (in /lib/libc.so.1)
==857==    by 0xFEF1446D: snprintf (in /lib/libc.so.1)
==857==    by 0x8050CC0: main (in /home/setup/bug)
==857==  Uninitialised value was created by a stack allocation
==857==    at 0xFEF1440C: snprintf (in /lib/libc.so.1)
==857==
```

Chapter 6

Conclusion

The thesis has described main directions and techniques in the program analysis area, together with representatives of actual tools. Basic capabilities of the tools have been presented and compared.

The main goal of this thesis, porting Valgrind to the Solaris/x86 platform, has been successfully reached. Many areas were studied for doing so, including internals of Valgrind, specific parts of the Solaris kernel, the runtime linker and the standard C library as well as the ELF format and the x86 processor architecture. In the case of the Solaris kernel, the focus was especially on understanding how programs are executed, how threads, signals and syscalls are implemented, and what individual syscalls do.

Based on this knowledge, a way to add support for Solaris/x86 to Valgrind has been devised and implemented. The necessary changes span over the complete Valgrind code base, with the main work done in the framework part (Coregrind).

The port is able to run many client programs. The programs can also make use of threads and signals. The port supports most important syscalls that are provided by the kernel, including file, socket and thread related syscalls. Out of eleven official tools, nine are supported. Only two thread error detectors (DRD and Helgrind) are currently not available. The functionality of the port was evaluated using the Valgrind test suite, proving that the port works correctly (except for a few minor problems).

Future work on this project will include enabling the two remaining tools, adding the core dump and gdbserver support, implementing more syscall wrappers and porting to the AMD64 architecture.

Appendix A

Installation and User Manual

The requirements to successfully build the port are: Solaris 11, GCC 4.x, Solaris ld, GNU autotools, GNU make, and Mercurial (for obtaining the source code). If all these are satisfied then the port can be built using the following commands:

```
$ hg clone https://bitbucket.org/setupji/valgrind-solaris
$ cd valgrind-solaris
$ ./autogen.sh
$ ./configure
$ make
$ make install
```

The port does not come with its own user manual. The official Valgrind manual should be used instead [57].

Appendix B

Contents of Enclosed CD

/	
valgrind-3.8.0.SVN.tar.gz	distribution package of the project
valgrind-solaris.patch	summary of differences between the original Valgrind and the port
DP_Pavlu_Petr_2012.pdf	thesis text
DP_Pavlu_Petr_2012.tar.gz	LaTeX source code of the thesis text

References

[1] Bessey, A.; Block, K.; Chelf, B.; etc.: A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, volume 53, no. 2, 2010: pp. 66–75.

[2] Bonwick, J.: The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, USENIX Association, 1994, pp. 6–6.

[3] Bonwick, J.; Adams, J.: Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, USENIX Association, 2001, pp. 15–33.

[4] Cantrill, B.; Shapiro, M.; Leventhal, A.: Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, USENIX Association, 2004, pp. 2–2.

[5] De Bus, B.; Chanet, D.; De Sutter, B.; etc.: The Design and Implementation of FIT: a Flexible Instrumentation Toolkit. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Foftware Tools and Engineering*, PASTE '04, ACM, 2004, pp. 29–34.

[6] Engler, D.; Chelf, B.; Chou, A.; etc.: Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, USENIX Association, 2000, pp. 1–1.

[7] Evans, D.; Guttag, J.; Horning, J.; etc.: LCLint: A Tool for Using Specifications to Check Code. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '94, ACM, 1994, pp. 87–96.

[8] Gregg, B.; Mauro, J.: *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD.* Prentice Hall, 2011, ISBN 0132091518.

[9] Hallem, S.; Chelf, B.; Xie, Y.; etc.: A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, ACM, 2002, pp. 69–82.

[10] Hastings, R.; Joyce, B.: Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, USENIX Association, 1991, pp. 125–136.

[11] Hunt, G.; Brubacher, D.: Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd Conference on USENIX Windows NT Symposium - Volume 3*, USENIX Association, 1999, pp. 14–14.

[12] Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C*. 2012. <http://download.intel.com/products/processor/manual/325462.pdf>

[13] Johnson, S.: Lint, a C Program Checker. *Computer Science Technical Report*, volume 65, 1978.

[14] Luk, C.; Cohn, R.; Muth, R.; etc.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, ACM, 2005, pp. 190–200.

[15] McDougall, R.; Mauro, J.; Gregg, B.: *Solaris (TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series)*. Prentice Hall, 2006, ISBN 0131568191.

[16] Nethercote, N.: *Dynamic Binary Analysis and Instrumentation*. Dissertation thesis, University of Cambridge, 2004.

[17] Nethercote, N.; Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, ACM, 2007, pp. 89–100.

[18] Oracle and/or its affiliates: *Linker and Libraries Guide*. 2011. <http://docs.oracle.com/cd/E23824_01/pdf/819-0690.pdf>

[19] Patil, H.; Fischer, C.: Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software: Practice and Experience*, volume 27, no. 1, 1997: pp. 87–110.

[20] Poletto, M.; Sarkar, V.: Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 21, no. 5, 1999: pp. 895–913.

[21] Serebryany, K.; Iskhodzhanov, T.: ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, ACM, 2009, pp. 62–71.

[22] Srivastava, A.; Eustace, A.: ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, ACM, 1994, pp. 196–205.

[23] Sun Microsystems, Inc.: *Stabs Interface*. 2004. <http://developers.sun.com/sunstudio/documentation/ss11/stabs.pdf>

[24] Traub, O.; Holloway, G.; Smith, M.: Quality and Speed in Linear-scan Register Allocation. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, ACM, 1998, pp. 142–151.

[25] Autoconf - GNU Project - Free Software Foundation (FSF). <http://www.gnu.org/software/autoconf/>

[26] Automake - GNU Project - Free Software Foundation (FSF). <http://www.gnu.org/software/automake/>

[27] A Comparison of Memory Allocators in Multiprocessors. <http://developers.sun.com/solaris/articles/multiproc/multiproc.html>

[28] Coverity Scan Site. <http://scan.coverity.com/>

[29] Coverity Static Analysis Data Sheet. <http://www.coverity.com/library/pdf/CoverityStaticAnalysis.pdf>

[30] Coverity Static Analysis Tools for C/C++, C#, and Java. <http://www.coverity.com/products/static-analysis.html>

[31] Description of the .PDB files and of the .DBG files. <http://support.microsoft.com/kb/121366/en-us>

[32] DUMA library. <http://duma.sourceforge.net/>

[33] Dwarf Home. <http://dwarfstd.org/>

[34] Dynamic program analysis – Wikipedia, The Free Encyclopedia. <http://en.wikipedia.org/wiki/Dynamic_program_analysis>

[35] FIT - the Flexible Instrumentation Toolkit. <http://www.elis.ugent.be/fit/>

[36] GDB Remote Serial Protocol. <http://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Protocol.html>

[37] GDB: The GNU Project Debugger. <http://www.gnu.org/software/gdb/>

[38] Ghosts of Unix past, part 3: Unfixable designs [LWN.net]. <http://lwn.net/Articles/414618/>

[39] GNU General Public License, version 2. <http://www.gnu.org/licenses/gpl-2.0.html>

[40] IBM Software - Rational Purify. <http://www-01.ibm.com/software/awdtools/purify/>

[41] List of tools for static code analysis – Wikipedia, The Free Encyclopedia. <http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis>

[42] The mpatrol library. <http://mpatrol.sourceforge.net/>

[43] Open Source Initiative OSI - Common Development and Distribution License (CDDL-1.0). <http://www.opensource.org/licenses/CDDL-1.0>

[44] Oracle Solaris Modular Debugger Guide. <http://docs.oracle.com/cd/E19963-01/html/817-2543/index.html>

[45] Oracle Solaris Studio. <http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html>

[46] Parallel Studio 2011 from Intel. <http://software.intel.com/en-us/articles/intel-parallel-studio-home/>

[47] PC-lint and FlexeLint Home Page. <http://www.gimpel.com/html/index.htm>

[48] Pin - A Dynamic Instrumentation Tool. <http://www.pintool.org/>

[49] Portable umem. <https://labs.omniti.com/labs/portableumem>

[50] A Quick Peek Under the Hood of Intel Parallel Inspector. <http://software.intel.com/en-us/articles/quick-peek-under-the-hood/>

[51] Splint Home Page. <http://www.splint.org/>

[52] STABS. <http://sourceware.org/gdb/current/onlinedocs/stabs.html>

[53] System V Application Binary Interface. <http://www.sco.com/developers/gabi/latest/contents.html>

[54] ThreadSanitizer - A Valgrind-based detector of data races. <http://code.google.com/p/data-race-test/wiki/ThreadSanitizer>

[55] Valgrind Home. <http://valgrind.org/>

[56] Valgrind port to FreeBSD/x86 and FreeBSD/AMD64. <https://bitbucket.org/stass/valgrind-freebsd>

[57] Valgrind User Manual. <http://valgrind.org/docs/manual/manual.html>

[58] What Is .SUNW_ldynsym? <https://blogs.oracle.com/ali/entry/what_is_sunw_ldynsym>

[59] WineHQ - Run Windows applications on Linux, BSD, Solaris and Mac OS X. <http://www.winehq.org/>

[60] Johnson, S.: Lint's source code. 1979. <http://minnie.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/lint>

[61] The IEEE and The Open Group: The Open Group Base Specifications Issue 7. <http://pubs.opengroup.org/onlinepubs/9699919799/>

Index