Valgrind for OSE

Lifei Tang

8/18/2012

Supervisor: Moris Behnam
Examiner: Thomas Nolte

Supervisor: Mathias Engan
Supervisor: Daniel Forsgren

Abstract

For programmers, it is always painful and hard to identify non-fatal errors like memory leaks, out of boundary errors or data race condition with traditional debug tools e.g. GDB. Today, there are many tools available to help the programmers to find these problems. The collection of Valgrind tools is a good example. Valgrind itself is an open source framework for debugging and profiling. It is today available for Linux, Darwin and Android, on hardware platforms such as ARM, x86 and PPC. Valgrind virtualizes the user mode environment and depends on the host OS environment. This thesis explores how Valgrind could be adapted to support an OSE (a real-time operation system product from ENEA software AB) target in a Linux host environment.

Acknowledgement

I would like to thank many people, without their support this work would not have been possible.

I am really grateful to my supervisor at ENEA Software AB, Mathias Engan, who offers tons of help for solving all kinds of issues from the technical problem to my the daily company life; and Daniel Forsgren, my special technical advisor at ENEA, who kindly and patiently helps me to harming the technical problems on a weekly basis and provide me lots of valuable ideas, without his helps, I won't even know how to start the whole thing.

I'd like to thank Dr. Moris Behnam and Prof. Thomas Nolte, my thesis supervisor and examiner at Mälardalen University, who provide me constructive advices on my thesis.

A special thanks for Fredrik Bredberg at ENEA Software AB, who give me clues that lead me out of the darkest period of time during the implementation.

Thanks for Sona Sarmadi, Martin Borg and Thomas Barnå at ENEA Software AB, who helps me out of environment setting up and utilization. I must thanks Dan Lilliehorn at ENEA software AB, who helps me to be accepted as a thesis worker at ENEA at the first place. Thanks to all the friends I have met at ENEA for creating an enjoyable working environment.

Last but not least, thanks to my friends and especially my parents, they are my strongest back up at any time.

Contents

Glossary

Notation	Description
Soft Kernel	SFK
OSE compact kernel	OSEck
Dynamic binary instrumentation	DBI
Dynamic binary analysis	DBA
Operation system	OS
Just-in-time	JIT
Intermediate representation	IR
Real-time operating system	RTOS
Basic block	BB
OSE Compact Kernel	OSEck
Board support package	BSP
Inter-process communication	IPC
Hardware	HW
Vg memory buffer	VB
General purpose register	GPR

1. Introduction

1.1 Execution Context

For the C/C++ like low level programing languages that leave the memory management to the programmers, protections against non-fatal errors e.g. memory leaks, are limited. As the current hardware and software systems become increasing complex, it is however, not an easy task for the programmer to identify these errors either by just running the program or by traditional debuggers that could be helpful only for finding the source of the fatal errors. Fortunately, there are already tools available in the market to handle these errors, for instance; IBM Rational PurifyPlus [8], Valgrind [4], BoundsChecker [9], Insure++ [10], etc.

Today, Enea software AB [5] – a global software and service company, supports Rational PurifyPlus for detecting programming mistakes in application under the OSE [11] Soft Kernel (SFK) environments for Solaris and Linux (OSE is one of the Enea's real time operation system product, while SFK is a simulation of OSE which allows OSE application runs on a host machine e.g. a Windows machine). This includes Rational Purify for detecting memory leaks and accesses outside allocated buffers' boundaries, Rational PureCoverage for code coverage and Rational Quantify for performance profiling. But Rational PurifyPlus is an expensive IBM commercial product, therefore it would be interesting to investigate an alternative - Valgrind in this case. However, is Valgrind a good candidate to replace Purify? The next section compares Valgrind with PurifyPlus in details, from which we make a conclusion that Valgrind is a qualified alternative.

1.2 Valgrind vs PurifyPlus

Valgrind is a Dynamic binary instrumentation (DBI) framework [1], which simplifies the creation of heavyweight dynamic binary analysis (DBA) tools - Valgrind tools in this case (Please refer to Chapter 2 for more details regarding how Valgrind and Valgrind tools works). The concept DBA refers to two different sub-concepts: (a) the first one is binary analysis which indicates the analysis is performed on the machine code thus making it language-independent but platform-specific [6]; (b) the second one is dynamic analysis, which usually refers to the tools that instrument the machine codes with analyze codes at the run time [6]. This analyze code is utilized to trace the execution to identify errors, it is run as a part of the original program which hardly disturbs the original program behavior except slowing down of the execution. While the term heavyweight here refers to the tools that needs to utilize complicate analyzing code and that need to track a lot of extra metadata [6].

While Rational PurifyPlus is an equivalent concept of Valgrind tools e.g. both of Rational Purify and Memcheck - a Valgrind Tool can detect memory access errors in program. A more detailed comparison between Purify and Valgrind followed:

1. Analysis category:

   Both Valgrind Tools and Rational PurifyPlus are DBA tools, to be specific, both of them works on executable code level which means that they are language-independent but platform-specific and no recompilation is required to analyze the code in normal cases.

2. Supported Platforms

   Table 1.1 shows the supported platforms by Valgrind and PurifyPlus.

   Table 1.1 Supported Platforms by Valgrind and PurifyPlus [4] [8]

Valgrind	PurifyPlus
x86, AMD64, PPC32, PPc64, ARM, S390X/Linux x86, AMD64/Darwin ARM/Android	(x86, AMD64, Intel 64)/Linux x86/Windows Intel 64, AMD64, Sun UltraSPARC/Solaris IBM POWER5, IBM POWEER6/AIX

3. Supported Functionalities

   As is shown in Table 1.2, both PurifyPlus and Valgrind Tools can detect similar memory errors, but Valgrind Tools can also support thread error detecting and heap profiling which could be quite useful in OSE/OSEck cases.

   Table 1.2 Supported Functionalities by Valgrind Tools and PurifyPlus [4] [8]

Feature	Valgrind Tools	PurifyPlus
Memory Analysis	· Memory leak detection · Uninitialized memory Access · Unallocated memory Access · Access outside allocated boundaries · Access beyond the stack pointer · Improperly free heap buffer	· Memory leak detection · Uninitialized memory Access · Unallocated memory Access · Access outside allocated boundaries · Access beyond the stack pointer · Access through null pointers
Profiling	· Cache and branch-prediction proliter · Call-graph generating cache and branch prediction profiler · Heap profiler	· Runtime performance profiler
Thread Errors	· Data Race · Dead lock · Misuse POSIX threads API · Inconsistent Lock Ordering · Lock Contention	
Extra	· Open source	· Identifies the lines and the functions in code that have not been exercised by testing .

4. User experience

   There are two aspects that might affect the user experience: user interface and speed.

   From user interface perspective, both PurifyPlus and Valgrind support command line and GUI interface, while PurifyPlus also is integrated with IDEs e.g. Microsoft Visual Studio and allows interaction with users at runtime which is very handy for many situations e.g. suppress an unwanted error at runtime.

   In terms of speed, Purify is also faster than Valgrind/Memcheck according to the speed benchmark on Appendix 3.

5. Cost

   User license price of Rational PurifyPlus for Linux and UNIX is USD $668,000 and the enterprise edition which support Windows and Unix application development cost even higher: USD $1,500,000 [8]. While Valgrind is open source, which means it is free and more flexible to be customized for supporting OSE.

After comparison, we can see that both Valgrind and PurifyPlus can detect similar memory errors, but Valgrind can also support thread error detecting and heap profiling. Meanwhile Valgrind is open source, namely it is free and more flexible to be customized to support analyzing OSE applications. Therefore we believe that Valgrind is a potential candidate to replacing PurifyPlus in terms of detecting errors of OSE application.

1.3 Thesis Target

Although Valgrind has rich ready-made Valgrind tools to detect all sorts of programs' errors, it is currently only available for Linux, Darwin and Android, consequently, one cannot directly use Valgrind to analyze OSE application. This thesis explores solutions of evolving a customized Valgrind to support the OSE target in a Linux host environment and implements a prototype to prove the feasibility of the solution.

1.4 Thesis Contribution

The contributions of the thesis are:

1. Propose and analyze different solutions to make Valgrind supports the OSE target for memory checking in a Linux host environment.
2. Implement a prototype to support two Valgrind tool plug-in: Memcheck and SGcheck (Both Memcheck and SGcheck are Valgrind tools that could detect memory errors) for detecting programming mistakes in applications under the OSE SFK environments, which includes:
   - Customize Valgrind core, Memcheck and SGcheck for OSE/OSEck use cases in terms of error suppression.

- Implement and analyze two different solutions of adding an extra layer (OSE-WRAPPER) of the OSE SFK to bridge the OSE SFK and Valgrind, but without modifying original SFK source code.
- Test cases and relevant documentations.

3. Investigate the Rational PurifyPlus and Valgrind and analyze the pros and con of both in detail.

1.5 Outline of report

- Chapter 2 gives the background knowledge of Valgrind and ENEA's RTOS products, which are necessary to understand the rest of the thesis.
- Chapter 3 describes the general problem of adapting Valgrind to support an OSE/OSEck target in a Linux host environment.
- Chapter 4 proposes and analyzes 4 possible solutions of of analyzing OSE/OSEck application through Valgrind tools.
- Chapter 5 describes the details of the implemented prototype.
- Chapter 6 concludes the whole thesis project and describes the future works.
- Reference provides a list of referenced literatures.
- Appendix 1 provides a list of modified Valgrind source code.
- Appendix 2 provides a list of system call that was wrapped and instrumented in OSE-WRAPPER.
- Appendix 3 describes the test cases and analyzes test result in detail.

2. Background knowledge

This chapter gives the background knowledge of Valgrind, OSE/OSEck and OSE/OSEck SFK which is necessary to understand the rest of the thesis.

2.1 Valgrind

Valgrind is a DBI framework, which makes it easy to build Valgrind Tools. The Valgrind tools are plug-ins to Valgrind's core: Valgrind core + tool plug-in = Valgrind tool [1]. A set of available Valgrind tools is introduced as follow [4]:

1. **Memcheck** is a memory error detector. It is capable to identify the memory errors like: memory leak, out of boundary errors and accessing uninitialized memory unit errors.
2. **Cachegrind** is a profiler that explores the interaction between the client program and host's cache hierarchy.
3. **Callgrind** is a profiler that is capable of simulating the cache behavior and branch prediction. In addition, it also generates a call-graph that records call history among functions in the client program.
4. **Helgrind** is a thread error detector that could find errors like deadlock, data race in multithread applications.
5. **DRD** is another thread error detector that serves similar purposes as Helgrind but with different analysis techniques.
6. **Massif** is a heap profiler that records the amount of heap memory used by an application.
7. **DHAT** is a heap profiler which tracks all the heap buffers allocated by client program and exams how they are used, e.g. the average lifetime of heap buffers, access ratios of heap buffers, etc.
8. **SGcheck** is a complementary tool to Memcheck. To be specific, SGcheck can detect overruns of stacks and global arrays which cannot be detected by Memcheck.
9. **BBV** is an experimental tool that records all the basic blocks the client program enters and the times of each basic blocks runs.

2.1.1 A simple Valgrind/Memcheck use case

The following is an example [4] of using Valgrind/Memcheck under Linux OS.

Assuming that we have an executable code called test.out whose source code is as following:

```
test.c:
1   #include <stdlib.h>
2   void test(void)
3   {
4           int* t = malloc(10 * sizeof(int));
5           t[10] = 0;        // problem 1: heap block overrun
6   }                         // problem 2: memory leak -- t not freed
7   int main(void)
8   {
9           test();
10          return 0;
11  }
```

To run the program under Valgrind/Memcheck, one need first install Valgrind properly, and compile the source code with *–g* option such that Valgrind could use the debug information of the target program, then using the following command line to run e.g. test.out under Valgrind/Memcheck:

```
valgrind --tool=memcheck --leak-check=yes ./test.out
```

The tool option specifies which Valgrind tool would be use, and the leak-check is a Memcheck option to enable memory leak check. The Valgrind output is as following:

```
==14964== Invalid write of size 4
==14964==    at 0x80483FF: test (test.c:6)
==14964==    by 0x8048411: main (test.c:11)
==14964==  Address 0x41c0050 is 0 bytes after a block of size 40 alloc'd
==14964==    at 0x4029ACC: malloc (vg_replace_malloc.c:263)
==14964==    by 0x80483F5: test (test.c:5)
==14964==    by 0x8048411: main (test.c:11)
==14964==
==14964== HEAP SUMMARY:
==14964==     in use at exit: 40 bytes in 1 blocks
==14964==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==14964==
==14964== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==14964==    at 0x4029ACC: malloc (vg_replace_malloc.c:263)
==14964==    by 0x80483F5: test (test.c:5)
==14964==    by 0x8048411: main (test.c:11)
==14964==
==14964== LEAK SUMMARY:
==14964==    definitely lost: 40 bytes in 1 blocks
==14964==    indirectly lost: 0 bytes in 0 blocks
==14964==      possibly lost: 0 bytes in 0 blocks
==14964==    still reachable: 0 bytes in 0 blocks
==14964==         suppressed: 0 bytes in 0 blocks
==14964== ERROR SUMMARY: 2 errors from 2 contexts
```

There are a lot of information in the output, one need to read them carefully, for instance, number "==14964==" in the example above indicate the PID; "40 bytes in 1 blocks are definitely lost in loss record 1 of 1" indicate there is memory leak error; "Invalid write of size 4" indicate there is a heap block overrun error. The error message not only specify the sources of the errors including the function name, source file name and line number, back trace of the function calls, but also might provide extra information e.g. in "Invalid write of size 4" error, extra information is given to indicate that the invalid written memory is just past the end of a block allocated with malloc() on line 5 of test.c.

Above is just a simple Memcheck use case, Memcheck can also detect errors like using uninitialized value, support more command line option to help users to identify the problems in the code [4].

2.1.2 Workflow of Valgrind Tool

Valgrind uses just-in-time (JIT) compilation techniques which includes dynamic binary recompilation and caching. To be specific, the client program is totally under Valgrind control and that none of the original programs get runs on the host platform[12]. Each Valgrind tool is a statically linked executable that contains both the tool code and the core code [1], which first grafts itself into the client process at start-up stage such that they could share the same address space, and then the Valgrind Tool recompiles the client's code, one basic block (BB) at a time. BB is a block of consecutive code, except the last instruction, no jump, exit or return instructions is allowed within a BB. The compilation procedure of one BB involves the following steps: (a) The Valgrind core disassembles the machine code into intermediate representation (IR) and then optimizes IR; (b) The optimized IR is then instrumented by the tool plug-in; (c) Finally, the Valgrind core transfers the instrumented IR back to host machine code which is called translations in this paper. The result translation is different from the original executable, in that it contains the extra instrument code added by Valgrind tools. The whole process of translating a BB is shown in the Figure 2.1 [1].

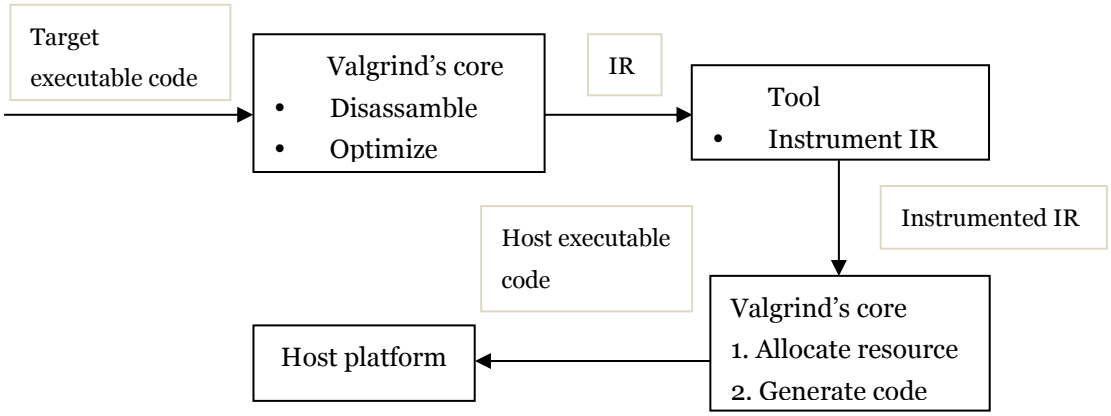

Figure 2.1 Translating a BB

To avoid repeating translating the same BB and speed up the performance, result translations

are stored in two tables that works as a two level code cache, specifically, the Valgrind core maintains a small direct-mapped table which acts as a cache for most recently-used translations and a full translation table which records all result translations, when the Valgrind core tries to generate the translation, it checks the direct-mapped table first, if it cannot find the corresponding translation, a full translation table check is made, if it still cannot find the corresponding code, a new translation is made and both the direct-mapped table and the full translation table are updated.

2.1.3 Shadow memory concept

The shadow memory concept [2] is not tied to Valgrind core, which is instead implemented on the tool side. The concept is introduced here at the beginning since it is generic to understand shadow memory tools like, Memcheck, Helgrind, SGcheck and the follow up contents in this paper.

Shadow memory provides a mean to record extra status information - shadow values for each memory unit that can be a register unit or an arbitrary user-mode memory address, while the kernel-mode address space is not considered since Valgrind works only in the user-mode. Supporting extra shadow values is complicated since it requires the tool plug-in to maintain two sets of data (both original and shadow values) for each memory unit without affecting execution, besides all the operations including reading, writing, allocation and de-allocation of memory units must be instrumented to inform the Valgrind tool and change the corresponding shadow values if necessary.

Table 2.1 shows how shadow values are recorded for each memory byte in the Memcheck tool plug-in: (a) Memcheck maintain a 'A' bit for each memory byte, which indicate the addressability. Specifically, 'A' bit is set as 1 if this memory byte is allocated and set as 0 if the memory byte is freed, such that Memcheck detects invalid memory access by checking each memory access. (b) Every memory byte is shadowed with 8 'V' bits, which indicate validity. Memcheck set the 'V' bit to 0 once the corresponding memory bit is allocated but not yet defined and set it as 1 if corresponding memory is written, thus reading undefined value errors are detected by checking each read memory unit action [7].

Table 2.1 how Memcheck record shadow values

A memory byte	XXXXXXXX	
Shadow Values	VVVVVVVV	A

2.1.4 Resource access protection

Because the client and tool are squeezed into a single process, many resources, such as, registers and memory need to be shared. To avoid resource conflicts, for instance, the client program modifies the register or memory that is used by Valgrind tool, the following techniques are used:

- Space-multiplexing [6] is used to divide the memory address space into three parts that are reserved for kernel, tool and client program respectively. Meanwhile all system calls related to resource allocation, such as, *malloc(), brk(), mmap(), mprotect()* are wrapped and checked to ensure the correct execution of memory space partition. For instance, the client try to allocate a memory area by mmap(), while this memory area is currently used by Valgrind tool, then Valgrind will force a failure to this memory allocation action.
- Time-multiplexing [6] is used to let the tool and the client program use the shared resources – the general purpose register (GPR) in this case, in the separate periods of time. To do that, a block of memory is maintained to keep all the client states – client's value for each GPR and the shadow state – the shadow values for each GPR, and the client states are loaded into real machine registers as necessary.

2.1.5 System call handling

Any system call that touches resource allocation or file descriptors are intercepted and checked to make sure that there will be no resource conflicts between Valgrind Tools and client program. Meanwhile, Valgrind works at user-mode, namely it cannot instrument kernel code and the system calls are executed un-instrumented. To ensure the system calls execute correctly, it is important to restore the client program states to the host register before calling the system calls, to be specific, the following steps should be taken:

1. Save the Valgrind tool's state to memory;
2. Restore the client state into the host registers;
3. Do the system call in client program;
4. Store the client state back to memory;
5. Restore the Valgrind tool's state.

2.1.6 Using Valgrind Core

This section only briefly describes three Valgrind features: function wrap, client request, error suppression, which are highly related to this thesis project. For more details of using Valgrind, please refer to Valgrind home page [4].

1. Function wrap

   Valgrind allows wrapping or even replacing any function in the client program, which is a very useful feature for examining or modifying the arguments and return value and instrumenting the client program at API level. The following is a simple example of using function wrap, assuming that we want to wrap a function called "*test*":

   int test (void) {return 0;}

   And one wants to instrument the *test* with an instruction that will print one line of "wrap_test start" into the terminal. Then the one can adopt following code:

```
int  I_WRAP_SONAME_FNNAME (NONE, wrap_test) (void)
{
        int  result;
        origFn  fn;
        VALGRIND_GET_ORIG_FN(fn);
        printf (“wrap_test started”);
        CALL_FN_W_v(result,fn)
        return  result;
}
```

In the code above, *int I_WRAP_SONAME_FNNAME (NONE, wrap_test) (void)* indicates that the original wrap_test function is wrapped. In the wrap function, *VALGRIND_GET_ORIG_FN (fn)* stores the address of original wrap_test function into variable *fn*, *printf (“wrap_test started”)* prints the message *“wrap_test start”* into the terminal, then *CALL_FN_W_v(result,fn)* calls the original *wrap_ose* function and store the return value into *result,* in the end the wrap function return the result into the client’s program.

2. Client request

Client request is a trapdoor mechanism implemented in Valgrind, via which the client program can communicate to Valgrind and the current tool. The following is a list of Valgrind and Memcheck client requests used in this thesis project [4]:

- *VALGRIND_MALLOCLIKE_BLOCK:* This request marks a region of memory as having been allocated by a *malloc()*-like function. For Memcheck (an illustrative case), it does two things (a) It records that the block has been allocated. This means that any addresses within the block mentioned in error messages will be identified as belonging to the block. It also means that if the block isn't freed it will be detected by the leak checker. (b) It marks the block as being accessible and undefined (if *'is_zeroed'* is not set), or accessible and defined (if *'is_zeroed'* is set). This controls how accesses to the block by the program are handled.
- *VALGRIND_FREELIKE_BLOCK:* This request is used in conjunction with *VALGRIND_MALLOCLIKE_BLOCK* to inform Valgrind that the allocated block is freed.
- *VALGRIND_CREATE_MEMPOOL(pool, rzB, is_zeroed):* This request create a memory pool for grouping memory chunks. The *pool* argument indicates the anchor address for a memory pool, the *rzB* argument specifies the size of the *redzones* placed around chunks and the is_zeroed argument indicates whether the pool's chunks are defined when allocated.
- *VALGRIND_DESTROY_MEMPOOL(pool)*: This request informs Memcheck that a pool is destroyed. Consequently, all records of memory chunks that associate to that pool are informed to be removed from Valgrind.

- *VALGRIND_MEMPOOL_ALLOC(pool, addr, size)*: This request informs Memcheck that a memory chunk is allocated and associate it with a specific memory pool. The address and the size of the allocated memory chunk is specified by arguments *addr* and *size*.
- VALGRIND_MEMPOOL_FREE(pool, addr): This request informs Memcheck that the memory chunk at *addr* is freed and should be disassociate with a memory pool.
- VALGRIND_MAKE_MEM_NOACCESS(_qzz_addr,_qzz_len): This request marks memory area from _qzz_addr to _qzz_len as "inaccessible".

3. Error suppression

   Valgrind Tools detect numerous problems even in system libraries and one is normally not in a position to fix them, certainly, one don't want to see these errors either (There can be quite a lot of them). Valgrind is capable of reading a list of errors to suppress at startup stage, and filter errors accordingly during error printing [4]. The following is an example:

```
{
    <_gconv_transform>
    Memcheck: Value4
    fun: _gconv_transform
    ...
    obj: /usr/X11R6/lib/libX11.so.6.2
}
```

   As is shown from the example above, each suppression usually has following components:

   - First line allows users to specify a customized name for the suppression, one can put any string inside the brackets.
   - Second line contains the name of the tool and the name of the suppression itself, which are separated by a colon.
   - Remaining lines contain the calling context for the error - a chain of functions and/or shared objects. The "…" in the example above is a wildcard, which means that any errors whose back trace begins with share object - "*usr/X11R6/lib/libX11.so.6.2*" and ends up with function – "*_gconv_transform*" could matches this suppression.
   - Finally, the entire suppression must be within a curly brace couple.

2.2 OSE/OSEck

Enea OSE is an embedded RTOS based on signal and microkernel architecture, while Enea OSE compact kernel (OSEck) is a DSP-optimized version of the Enea OSE RTOS, which

is suitable for high performance, memory constrained applications [11]. Similar as the other RTOS products, OSE/OSEck has the common RTOS features: portable, scalable, preemptive, multi-tasking, deterministic, robust and reliable. However, OSE is also unique due to its microkernel architecture and its prime inter-process communication (IPC) mechanism-direct message passing.

It wouldn't be possible and necessary go through all the details of OSE/OSEck in this section, only the concepts that highly relevant to this thesis project are mentioned in the following text, which including the OSE Architecture in general, concept of process and signal, heap memory and how OSE handle the error. No particular introduction for OSEck since as a slim version of OSE, they share the features in most cases.

2.2.1 Architecture

As is mentioned before, OSE is based on microkernel architecture [13] which means that many services normally designed as a part of the operating system kernel can be, and are, implemented as plugins and applications. OSE also has a layered architecture which can be viewed as sets of functionalities running on top of each other. Specifically, in OSE, the board support package (BSP) sits at the bottom layer, upon which the micro kernel is built and provides services for the rest of the kernel layers [11].

OSE is a signal based RTOS (the signal concept in OSE is different from the UNIX signal in that it is more like a message containing data which is queued in the process' signal queue read with a receive system call and do not trigger any synchronous signal handler), not only because the prime IPC in OSE is signal transferring, but also the way OSE performs the system call. To be specific, the OSE kernel creates several kernel service processes during the system boot up, when the application call the system call, instead of a standard function call after trap, the OSE kernel in turn sends a predefined signal to a certain kernel service process that performs the functionality of that system call.

2.2.2 Process

Processes are the most fundamental building blocks in OSE. The process concept here corresponds to light-weight processes, threads and tasks in other operation systems.

2.2.3 Domains, Blocks, Pools and Heaps

Processes can be grouped together into blocks consisting of one or several processes. The block abstraction is efficient for grouping related processes together to form a subsystem. Processes in the same block share the same memory pool and heap, from which the signal buffer and heap buffer are allocated respectively.

Domains are memory areas that are protected from each other by using a memory management unit (MMU), each domain can contain one or multiple blocks and in turns

contains one or multiple pools and heaps. The Figure 2.1 is an example to demonstrate the relationship between domains, blocks, heaps and pools.

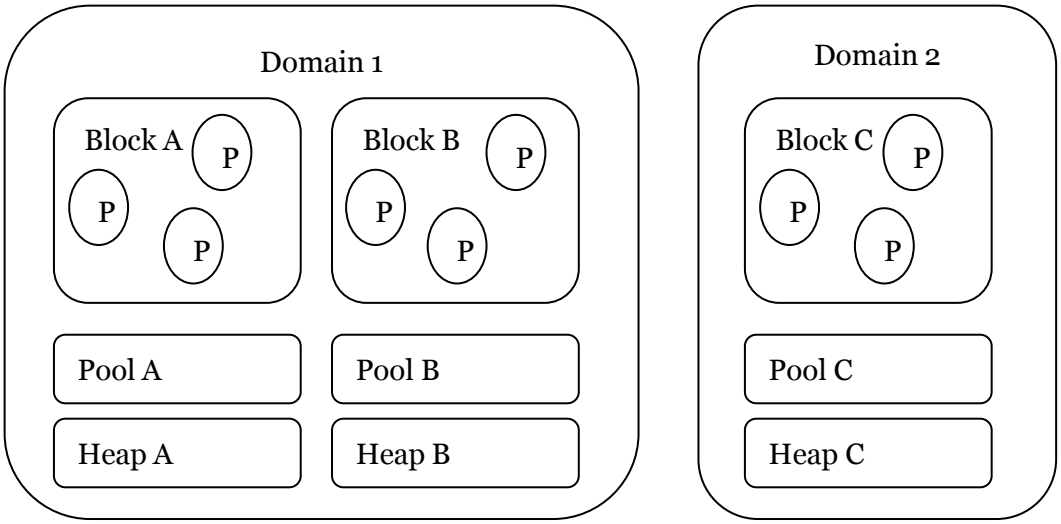

Figure 2.1 Processes in Blocks with Pools in Domains

2.2.4 Signal

In OSE there are several available ways for communication and synchronization between processes, including signals, semaphores, fast semaphores, or mutexes and among them, using signals is the recommend way of communication between two processes.

- **Signal buffer**
  A signal as it is shown in Figure 2.1 is a message buffer that is sent from one process to another, which contains "Administer Block", "Signal Number", "Data", and a one byte "Endmark". The "Administer Block" is used to record signal attributes including which process that sent the signal (the sender), which process it was sent to (the addressee), which process owns the signal (the owner), signal size, etc. The "Signal no" is used by a receiver to identify what the signal is about and what data it contains. The "Data" area records the content of the signal. Finally the "Endmark" is one byte filled with "0xEE" and is examined by the error handler if it is overwritten.

Administer block	Signal number	Data	Endmark

Figure 2.2 Signal buffer structure

- **Signal Passing**
  To pass a signal, the sender process must allocate a signal buffer with a proper size in its own pool, for example, in Figure 2.1, processes in block A can only allocate signal buffer from pool A. If signals are transferred between two processes reside in the same domain, the sending process only copies a pointer to a signal buffer in its pool. The receiving process then uses this pointer to access the signal buffer. Communicating processes can access signal buffers in all pools in their domain, but they can only allocate buffers in their own pool. If a signal is sent between processes

that use pools located in different domains, the signal buffer will be copied to the receiving pool. This is required, because the receiving process only has access to the pools within its own domain.

2.2.5 Private and Shared Heap Buffer

Heap buffer is a memory area allocated from heap through ANSI heap calls or OSE specific heap calls. In OSE, heap buffer can be divided into private heap buffer and shared heap buffer. The private buffers are by default owned by the process that allocated them, and cannot be accessed by the other processes, while the shared buffer can be allocated, deleted or access by all processes in the program. Another significant difference between the private and shared memory is that the private heap buffers are automatically garbage collected, specifically, when one process is killed, all the private heap buffer owned by this process are freed automatically, while as shared heap buffer stays allocated until the block (program) is deleted.

2.2.6 Limitation of Internal Error Checks in OSE kernel

In order to get information about incorrect usage of the operating system as early as possible, OSE is capable of performing extensive error checking at runtime, this includes parameter checks, validating the caller, buffer and stack checks, initialization checks, etc. When an error is detected, the corresponding error handler is invoked, which can define the backbone of the error recover policy and help the user to find the source of the error.

Although the OSE error checking mechanism is powerful in most cases, it still has its limitation in memory check cases. In OSE, buffers and stacks have an endmark that enable several system call to detect whether a buffer or stack has been overwritten beyond the boundary, however, if there is a pointer accessing the memory address without modifying the endmark, there is no chance for OSE to detect the error. Even if OSE detects the out of boundary error when an endmark is overwritten, it can only report the overwritten endmark address which is not very helpful for the user to find the source of the error. In addition, OSE is not capable of detecting memory leak errors and access of uninitialized value. These are as well the reasons why we resort to other tools like National PurifyPlus or Valgrind to help OSE to identify the memory errors. Please go to Appendix 3 for an example to show the limitation of OSE error checks.

2.2.7 OSE BIOS

The Bios concept in OSE has nothing to do with the Basic Input/Output System (BIOS) term in the desktop PC world, it offers basic service of using System call/Trap/Software Interrupt Mechanism. Whenever an application executes a system call, the execution jumps to the BIOS component, which allows programs to call the executive or other components that have registered with Bios, without knowing the address of the executive or component. It also

provides a way for the user mode code to switch to the supervisor mode in a controlled way.

2.2.8 OSE/OSEck SFK

The OSE/OSEck SFK runs as a single host process on a host OS (Windows, Solaris or Linux) and mimics the behavior of OSE/OSEck, such that one can run the OSE application on a host. OSE/OSEck SFK is like any other OSE/OSEck targets except a few differences, however, this section only mentions three of them which are highly relate to this thesis:

1. The SFK uses virtual interrupts, interrupts are simulated by polling.
2. In order to simulate OSE processes, threads are used in the host.
3. To simulate the memory management of other OSE/OSEck targets, SFK first use standard malloc() or mmap() to allocate a huge memory area from the host, from which the OSE customized resource allocation functions allocate buffers.

3. Problems and Possible Solutions

This chapter introduces the problems of applying Valgrind tools to analyze OSE application, and then presents and analyzes 4 different potential solution.

3.1 Problems

Valgrind works on user mode and doesn't trace into system calls, but it needs to intercept system calls to understand and monitor system behavior. For instance, Valgrind intercepts all system calls that touch memory resource allocation/deallocation. Specifically, these system calls should be wrapped and checked to avoid resource conflicts, and for Valgrind tools that using shadow memory concept, these system calls should be instrumented to update the shadow value of memory units. Valgrind also need to intercept the standard dynamic loader system calls like dlopen() and dlclose(), otherwise Valgrind won't be informed when the dynamic loaded modules are executed .

Although the recent version of Valgrind is fairly robust and is capable of running on its own without relying on external library, even the standard C library [1], it is still OS platform specific. Valgrind is OS specific since the system call it needs to intercept is OS specific. Valgrind currently supports Linux, Darwin and Android, such that it is able to capture the corresponding system calls in these three platforms. But when it comes to OSE, it used customized resource allocation/deallocation and dynamic loader system calls that are not supported by Valgrind, consequently, one cannot using Valgrind directly to analyze OSE application.

3.2 Possible solutions

Here 4 possible solutions are presented regarding analyzing OSE/OSEck application through Valgrind tools, especially the shadow memory tools.

1. Solution 1 (S1) - OSE/OSEck software kernel (SFK) under Valgrind
2. Solution 2 (S2) - OSE/OSEck under Valgrind
3. Solution 3 (S3) - OSE/OSEck process under Valgrind
4. Solution 4 (S4) - Port Valgrind to OSE

Problems of each solution might encounter and the corresponding ways of solving these problems are analyzed. Meanwhile, in this chapter, these solutions are evaluated in terms of efforts needed to achieve the result, applicability and the accuracy of analyzing results.

3.2.1 OSE/OSEck software kernel (SFK) under Valgrind

As it is shown in Figure 3.1, the idea behind S1 is treating OSE/OSEck SFK and the OSE/OSEck application together as a normal Linux application that run under the control of Valgrind, while Valgrind, OSE/OSEck SFK and the target application runs within a normal Linux process. In addition, Linux and any HW that supports Linux can be the host OS and host hardware (HW) in this solution.

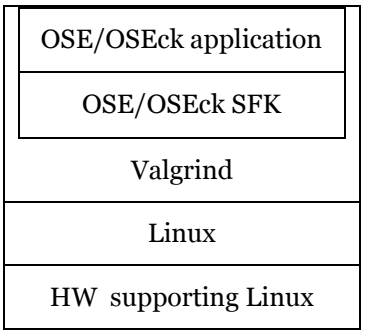

Figure 3.1 OSE/OSEck SFK under Valgrind

3.2.1.1 Problems of S1

There are essentially four problems need to be handled in S1:

The first problem is caused by the memory management mechanism of OSE/OSEck SFK. Valgrind tools like Memcheck, Massif and Hegrind, rely on knowing the border of the buffer when one is allocated, which can be automatically done by the Valgrind for standard allocated functions: malloc(), calloc(), realloc(), memalign(), new, new[], free(), delete, delete[]. But when it comes to OSE/OSEck SFK, it has its own resource allocation functions such as alloc() for resource management. In this case, Valgrind tool cannot be informed properly regarding the allocation of the memory block and in turn fail to detect, for instance, heap block overrun and memory leak problems. Therefore, a mechanism of informing Valgrind is necessary when OSE/OSEck SFK tries to allocate memory resources. A simple solution to solve this problem is to first intercept all the customized resource allocation functions by function wrap mechanism provided by Valgrind, then instrument them by Client Request - another Valgrind built-in communication mechanism between client program and Valgrind Core, to pass messages to inform Valgrind core about all the necessary memory allocation/de-allocation information.

The second problem is regarding the resource management mechanism that applied in SFK, which first uses standard malloc() or mmap() to allocate a huge memory area, from which the OSE customized resource allocation functions allocate buffers. Valgrind automatically detects and marks the memory area allocated by standard malloc() and mmap() as “accessible”, as a result legalizing all the memory access inside this memory area. A simple solution would be marking the whole area as “inaccessible” so that Valgrind can detect illegal memory access.

The third problem is about the load module, which is an executable file that can be loaded by OSE SFK at run time. Valgrind is capable of detecting and handling the standard programming interfaces to dynamic linking loader - dlopen(), dlcose(), etc. however instead of using standard dlopen(), a customized dynamic loader is implemented in OSE, which cannot be detected by Valgrind. As a result, the load module would be executed un-translated, and Valgrind would not detect any errors from load modules. Three solutions are proposed here to solve this problem:

1. The first one would be OSE BIOS level interception of the customized resource allocation functions. Although the load module may be executed un-translated, all the resource allocation functions are system calls that would in turn jump to BIOS components. If the BIOS components that are part of OSE kernel are instrumented properly, all the customized resource allocation would be detected. However Valgrind still cannot read the debug information from the load module. As a result, Valgrind cannot display the source of errors properly. To solve this problem, an extra piece of post-process program is necessary to help Valgrind display the errors.
2. The second possible solution would be intercepting the OSE run time loader interface and implement a customized user request to inform Valgrind regarding which executable file is loaded. Meanwhile extra functionality should be implemented in the Valgrind core to handle the new client request.
3. The third solution is straight forward, instead of compiling a module as a load module, compile it as part of OSE SFK kernel. After passing the Valgrind tools' memory checks, it would be safe to use it as a load module in the real product.

The forth problem is regarding error suppression. SFK runs under Valgrind, Valgrind reports thousands errors of the SFK, which could be either true errors or false positives. But in either case, these errors should be suppressed for analyzing an OSE application, since the end-user is not in the position to fix the errors that originate from SFK. To solve this problem, certain mechanism of suppressing errors should be introduced, this can be solved by modifying Valgrind suppression mechanism or add an extra post-processing software to filter the misreported errors.

3.2.1.2 S1 Evaluation

This section evaluates the S1 in terms of the amount of efforts that needs to achieve S1, result accuracy and the applicability.

1. **Relative Small Effort Needed to Achieve S1**

   Compared with the other solutions that will be introduced later in this chapter, achieving S1 needs relatively small efforts. In S1, the following mechanisms need to be implemented:

   - Informing Valgrind when SKF allocate memory resources.
   - Handling load module.
   - Filtering errors originated from OSE core.

2. Debatable Result Accuracy

If the host HW is same as or compatible to the target hardware, a more reliable analyzing result could be achieved. But considering that x86 is widely used as the host platform, which can be, in most cases, different from the actual target machine in embedded industry. As a result, instead of analyzing the machine code for the target platform (like PPC, MIPS or ARM) that can be distinct with x86 architecture in terms of endianness, alignment and variable size etc., Valgrind is actually profiling the application binary for X86. Therefore the accuracy of the analyzing result is under no guarantee in some cases. Meanwhile, S1 inherits the all the difference between hard kernel and soft kernel, for instance, the soft kernel only supports memory model 1, which means that the soft kernel always copies the signal buffers between OSE processes - an OSE core library flavor with protection is needed. The OSE/OSEck target can use a more efficient memory model to share memory for buffers and only transfer signal buffers using pointers.

3. Good Applicability

However, S1 is very applicable. S1 support profiling both OSE and OSEck application, meanwhile S1 support any host HW that support Linux.

3.2.2 OSE/OSEck under Valgrind

As is shown in Figure 3.2, compared with S1, S2 introduces OSE/OSEck instead of OSE/OSEck SFK that runs under Valgrind. In addition, S2 uses target hardware as the host machine. However, as in S1, Linux is the host OS in S2, while Valgrind, OSE/OSEck and the target application runs within a normal Linux process; besides, it is also holds that both OSE/OSEck and OSE/OSEck application are under Valgrind control.

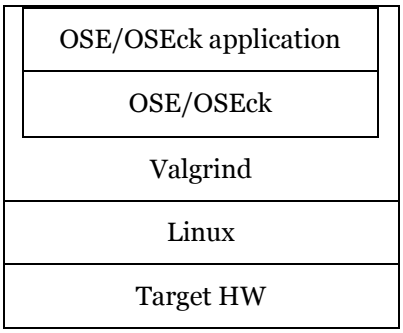

Figure 3.2 OSEck under Valgrind

3.2.2.1 Problems of S2

There are 5 problems that need to be considered in S2:

The first problem: Valgrind runs in user mode while OSE/OSEck kernel run in supervisor mode, therefore how to make OSE/OSEck run on the user mode is the key of S2. There are two approaches to achieve this solution:

- Modify Valgrind:

In this case, the modified Valgrind need to intercept the supervisor mode instruction in OSE/OSEck and replace it with user mode instructions.

- Modify OSE/OSEck:
  In this case, directly modify OSEck to make it run in user mode.

The second problem is about interrupts and system clock. A solution adopted by the SFK may also be applied here. To be specific, SFK do not support real interrupt, instead, SFK uses virtual interrupts: polling to find out whether an interrupt has occurred, then the SFK invokes the corresponding interrupt process. The system call is then simulated by a virtual interrupt.

The third problem also occurs in S1. OSE/OSEck use customized resource allocation functions which are not supported by Valgrind. Despite the implementation details, the solution is essentially the same as S1.

The forth problem is about avoiding resource confliction, unlike S1 – under the SFK environment, the OSE SFK first allocate a huge memory area by standard malloc() or mmap() system call, from which the OSE customized resource allocation functions allocate buffers. And Valgrind is capable of automatically detecting and checking standard malloc() or mmap() system calls to make sure the memory area allocated by OSE SFK won't conflict with the Valgrind address space. But in S2, the OSE/OSEck's customized resource allocation system calls will directly allocate memory from PC. Such that a mechanism should be implement in Valgrind to detect and check these OSE/OSEck resource allocation system calls to avoid resource conflict.

The fifth problem is also encountered in S1, the load module issue, and the solution would be the same as well.

The sixth problem, is also relate to error suppression. In this solution OSE/OSEck is under Valgrind control, Valgrind reports all the core errors from OSE/OSEck. It is again a similar problem in S1 and the solution would be as the same.

3.2.2.2 S2 evaluation

This section evaluates the S2 in terms of the amount of efforts that needs to achieve S2, result accuracy and the applicability.

1. **Relative High Effort Needed to Achieve S2**
   S2 requires either that all the OSE/OSEck supervisor mode instructions are intercepted or that the whole OSE/OSEck is ported into user mode. Also the virtual interrupt simulation that is used in the SFK should also be implemented here. In addition, S2 inherits most of the problems that were encountered in S1, therefore the workload of S2 is believed to be heavier compared to S1.

2. **High Result Accuracy**
   Since the host platform is the same as the target platform, customers can directly use

Valgrind to profile the binary of an OSE/OSEck application for a target platform, a very accurate analyzing result is then expected in S2.

3. Restrict Applicability

S2 requires the target HW supporting Linux, while OSEck is usually applied in resources constrained target platform which in some cases cannot support Linux, the applicability of S2 is somehow restricted from this perspective.

3.2.3 OSE/OSEck process under Valgrind

Valgrind tends to report errors originated form OSE/OSEck core which is a common problem that occurs in both S1 and S2. S3 aims at solving this problem fundamentally by only allowing Valgrind instruments the application code. As is shown in Fig 3.3, only one OSE/OSEck process runs under Valgrind, while the OSE/OSEck simulation is a layer under Linux but outside the control of Valgrind, thus Valgrind is only capable of instrumenting an OSE/OSEck process. Once the OSE/OSEck process call a system call, Valgrind is in charge of forwarding this call to the OSE/OSEck simulation lib and correctly returning the result to the OSE/OSEck process. In addition, it is possible to run the whole OSE/OSEck application under Valgrind, which requires integrating the scheduler of OSE/OSEck into the Valgrind core.

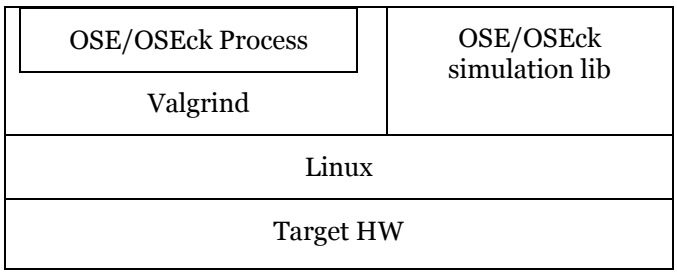

Figure 3.3 OSE/OSEck process under Valgrind

3.2.3.1 Problems of S3

There are three problems that need to be considered in S3:

The first problem is regarding implementing an OSE/OSEck simulation layer under Linux. One possible way is using a cutting version of OSE/OSEck SFK to serve this purpose.

The second problem is about how to handle the OSE/OSEck system call: in S3, OSE/OSEck is not under Valgrind control while the OSE/OSEck application is, therefore all the system calls should be intercepted and instrumented with customized client requests, thus Valgrind can forward system call to OSE/OSEck simulation lib, return the result to the client OSE/OSEck application and handle OSE/OSEck resource allocation/deallcation system calls properly. Meanwhile, all the OSE/OSEck resource allocation system calls should be checked to avoid resource conflicts.

The third problem is related to interrupt and system clock issue, a problem also occurs in S2.

Thefore the solution is also the same as in S2: adopting virtual interrupts and then simulating system calls by virtual interrupts.

3.2.3.2 S3 evaluation

This section evaluates the S3 in terms of the amount of efforts that needs to achieve S3, result accuracy and the applicability.

1. **Relatively High Effort Needed to Achieve S3**
   S4 requires relative heavy workload. First, new client requests needs to be implemented and the Valgrind core has to be modified correspondingly to support the new client requests. Second, a OSE/OSEck simulation lib need to be implement and all the OSE/OSEck system calls must be intercepted and handled properly. Third, like S2, the virtual interrupt simulation that used in SFK should also be implemented in S3. Finally, if one would like to extend S3 to profiling the whole OSE/OSEck application more work load is required to integrate the OSE/OSEck scheduler into the Valgrind core. But for the test purpose, a process that only uses limited number of system calls can be used, thus only a small amount of system calls need to be intercepted and simulated.

2. **Relative High Result Accuracy**
   Similar to S2, the host machine is the same as the target platform in S3, customers can directly use Valgrind to profile the binary of a OSE/OSEck process for a target platform, thus a high accuracy profiling result should be achieved for a single process OSE/OSEck application. Meanwhile, in S3, Valgrind only instruments the OSE/OSEck application and the errors that originated from kernel won't be reported. However, S3 can only support one process applications, or just profile one process of a multi-process application, in later case, the accuracy of the profiling result is debatable.

3. **Restrict Applicability**
   As in S2, S3 also requires the target HW supporting Linux, while OSEck is usually applied in resources constrained target platform which in some cases cannot support Linux, the applicability of S3 is somehow restricted from this perspective. Besides, S3 can receive accurate profiling result for single process application, which may not hold on multi-process cases.

3.2.4 Port Valgrind to OSE

As it is shown in Fig 3.4, in this case Target HW and OSE are the host platform and the host OS respectively and Valgrind is ported into OSE such that the binary of an OSE application is under Valgrind control.

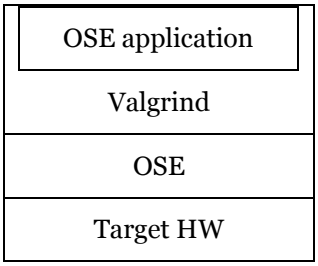

Figure 3.4 Port Valgrind to OSE

3.2.4.1 Problems of S4

The problem of this solution is that how to properly port Valgrind into OSE and at least the following points should be considered:

- The libraries and APIs that are used by Valgrind but not supported by OSE should be replaced.
- New code for handling address spaces is necessary.
- System calls that involve resource management should be wrapped or replaced properly to make sure that there are no resource conflicts between Valgrind and the client program.

3.2.4.2 S4 evaluation

This section evaluates the S4 in terms of the amount of efforts that needs to achieve S4, result accuracy and the applicability.

1. **Very High Efforts Needed to Achieve S4**
   Port Valgrind to OSE requires high amount of modification, according to the problems mentioned in S4.1.

2. **High Result Accuracy**
   Similar to S2 and S3, the host machine is the same as the target platform in S4, customers can directly use Valgrind to profile the binary of an OSE process for a target platform, therefore, a high accuracy profiling result should be achieved. In addition, since that Valgrind works in user mode and cannot trace into OSE system call, the misreporting problems that may occur in S1 and S2 is avoided.

3. **Restrict Applicability**
   S4 might be a good idea for OSE, but maybe not that good for OSEck which is usually target for resource constrained hardware, while Valgrind need relative large memory foot-print support.

3.2.3 Solution Tradeoff

From the analysis above, we can see that S1 is probably the easiest solution to be achieved and also most applicable one compared to the others, but its analysis result may be not that

reliable in some rare cases if the host machine is different from the target. While the S2 and S3 can reach a more accurate result since the solution requires that the host platform and the target should be the same, in addition, the second and third solution needs more effort to be achieved and less applicable compare with S1. Comparing with other 3 solutions, S4 requires most works to be achieved, although it can reach a reliable result but the applicability is also low in S4.

From the overall perspective, S1 is the best solution and in the next chapter, a prototype will be introduced, which is based on S1 and supports two Valgrind tool plug-in: Memcheck and SGcheck for detecting memory errors in OSE application.

4. Prototype Implementation and Result

This chapter presents a prototype that supports two Valgrind tool plug-in: Memcheck and SGcheck for detecting memory errors in applications under the OSE SFK environments. Under little modification, this prototype is supposed to support debugging OSEck application as well, since most interfaces of OSE and OSEck are the same, however, due to the time limitation, this prototype currently supports debugging the OSE application only.

4.1 Challenges of prototype implementation

As is described in Chapter 3, there are mainly three problems to handle to involve Valgrind tools to detect memory errors in OSE application under SFK environments:

1. How to apply the shadow memory concept for detecting memory errors. To be specific, how to inform Valgrind to update the shadow value of memory units when OSE customized resource allocation/deallocation interfaces are used.
2. How to only report errors that originated from OSE application, namely a solution to suppress all the OSE SFK core errors is needed.
3. How to handle the load module, which uses the customized dynamic loader interface.

To handle these problems, Valgrind source code is modified to suppress all the OSE SFK core errors; meanwhile OSE-WRAPPER, an extra lay of OSE SFK to bridge OSE SFK and Valgrind, is implemented.

4.2 Customized Valgrind for OSE

If one runs the OSE SFK with the original Valgrind Tools like Memchek, SGcheck, thousands of core errors will be reported and the situation could be much worse when OSE-WRAPPER is introduced, the reported core errors could be more than half billion from experiences. These errors could be either real errors or false positives, but in terms of debugging OSE application, the users should not be disturbed from these errors in either case. This section presents a solution to customize Valgrind to only report errors that relates to the user application.

4.2.1 Limited Valgrind build-in suppression mechanism

The Valgrind built-in suppression mechanism is handy in most cases, but not well enough for debugging OSE application. As is described in section 2.1.9, to successfully suppress an error,

Valgrind requires the user specifying a sequence of functions' names in a suppression, which should exactly match the back trace of the error targeted to be suppressed. However, before actually run the client program, one cannot predict the exact back trace of target error that need to be suppressed. We hope that there is a more robust suppression type, which, for instance, contains a list function names from the OSE user application, such that Valgrind only report the errors that relates to the functions specified in the list.

4.2.2 Solution analysis

One possible way to solve the problem is extending the existing Valgrind error suppression mechanism by implementing new error suppression types in the Valgrind core. Another alternative is adding a post-process software to redirect and filter the output of Valgrind, such that only desired error messages are printed to users. For the performance reason, the first solution is adopted. The reason why directly modifying Valgrind can improve the performance is as following:

In Valgrind, all the suppressions are stored in a link list and all the recorded errors have to go through the entire list, such that only errors that cannot match any suppression in the list are reported to the user. However, this suppression list tends to be large, which at least includes hundreds of the Valgrind predefined suppressions that used to suppress the false positive errors from the standard library. This list can be huger if there is a large number of user specified suppressions. Consequently, it could be time consuming e.g. half million core errors that would be suppressed later anyway, go through the whole suppression list to match suppressions. Therefore, it would be idea if there is a separated stored suppression type "A", which, for instance, specifying all the names of target functions from which Valgrind reports the errors. Then instead of directly going through the whole suppression list, each error try to match with suppression type "A" first, such that only the errors that mismatch with suppression "A" would have the chance to go through the entire suppression list, which is in most cases a very small number of errors compared with e.g. half million core errors. As a result, the overall performance is improved.

4.2.3 Implementation of Customized Valgrind

Three customized suppression types including: "Check", "UnCheck" and "OseLeakFilter", are added into Valgrind core to support Memcheck and SGcheck debugging OSE applications. Instead of modifying Valgrind tool, the Valgrind core is directly modified to support new suppression types since these suppression types can still be useful when the other Valgrind tools are needed to be customized for OSE/OSEck.

- **"Check" and "Uncheck "suppression type**

  "Check" and "Uncheck" suppression type are added to make Valgrind avoid reporting the errors that originated from OSE SFK core by matching the function names in the back trace of the error with a chain of function names that specified by "Check" or

"UnCheck" suppression type. The "Check" suppression type is recommended to use, which is a faster way of filtering the core errors. The following is an example of using "Check" suppression type:

```
{
    <insert_a_suppression_name_here>
    ose:Check
    fun:hello_main
    fun:world
}
```

The function list above informs Valgrind that only reports the errors that relate to the function names specified in the "Check" suppression type. For instance, in the example above, Valgrind Tools only report the error whose back trace contains hello_main function or world function. For analyzing the OSE application cases, one only needs to specify the process names, instead of names of all the functions that used in application and the sequence of process names that specified in "Check" would not affect the suppression result. Meanwhile it is perfectly fine that only specifying names of processes that one is interested to investigate in the "Check" suppression and the accuracy of memory analyzing results won't be affected either. For instance, if one only wants to see the errors that relate to "world" function, the example above can be modified as following:

```
{
    <insert_a_suppression_name_here>
    ose:Check
    fun:world
}
```

Valgrind stores suppression's function list in an array. Namely the size of the function list needs to be statically specified in the code (or in the other word, hard-coded). The size of first loaded suppression is specified by VG_MAX_SUPP_CALLERS_OSE in m_errormgr.c file which is preset as 200, for the later loaded suppressions, the size is specified by VG_MAX_SUPP_CALLERS which is preset as size 26. It would not be a good idea to set them equally as 200 or even larger which may lead to out of stack size issue. Therefore it is reasonable that the "Check" or "UnCheck" suppression must be added first in suppression file. Such that if one need to use "UnCheck" or "Check" suppression type one can tweak VG_MAX_SUPP_CALLERS_OSE for different cases but remain the VG_MAX_SUPP_CALLER constant. If one try to use "Check" or "UnCheck" but fail to locate them in the first place of the suppression file, Valgrind reports a fatal error and exits, which also means that "Check" and "UnCheck" are not allowed to use at same time!

If one replaces the ose:Check with ose:UnCheck in the example above, then all the function names of OSE system calls need to be specified in these suppression to inform Valgrind that if an error only relate to OSE system calls, suppress this error.

However, the number of OSE system calls is considerably huge, namely the suppression match procedure is time consuming and a huge value must be given to VG_MAX_SUPP_CALLERS_OSE which in turn consumes a lot of stack space, therefore "Check" suppression is recommended to use.

- **OseLeakFilter suppression type**

  "OseLeakFilter" suppression type is created to hide the OSE-WRAPPER layer from the end users. Valgrind trends to back trace the error into functions that specialized for OSE-WRAPPER. These functions (usually beginning with vg_) belong neither to OSE interface nor to functions specified by user applications, therefore, it would be confusing if they appear in error reports. For instance, without using "OseLeakFilter" suppression type, one may get a memory leak error from Valgrind/Memcheck like the following:

  ```
  ==30101==  at 0x893D007:  malloc(vg_replace_malloc.c:263)
  ==30101==  by 0x877B667:  vg_malloc(ose_wrapper.c:1224)
  ==30101==  by 0x8588776:  vg_create_buf(ose_wrapper.c:1216)
  ==30101==  by 0x818A2EF: heap_alloc (heapapi.c:486)
  ==30101==  by 0x806433F: hello_main (hello.c:114)
  ==30101==  by 0x818F94A: ose_shell_child (addrem.c:118)
  ==30101==  by 0x8105440: process_setup (targint.c:1508)
  ==30101==  by 0x4E2835A: start_thread (in /lib/libpthread-2.4.so)
  ==30101==  by 0x4D80C0D: clone (in /lib/libc-2.4.so)
  ```

  From the example above, the function *"hello_main"* is from user application, while *"heap_alloc", "ose_share_child", "process_setup"* and "*start_thread*" are from OSE, and *"clone"* is standard Linux API. Therefore, users understand where these functions come from. However, the "*vg_malloc*" and "*vg_create_buf*" are from the OSE-WRAPPER, which is not open to the user and can be fairly confusing. The "OseLeakFilte" suppression is created to filter these functions in the error back trace, to be specific, once a function name in an error back trace matches any function name that specified in "OseLeakFilter" suppression, the error back trace stopped. The "OseLeakFilter" suppression is specified in default.supp, which should not be modified by user in normal cases. Now back to the example without using "OseLeakFilter" again, while this time "OseLeakFilter" is specified properly in default.supp, the analyzing result is as follow:

  ```
  ==30111==  at  0x818A2EF: heap_alloc (heapapi.c:486)
  ==30111==  by 0x806433F: hello_main (hello.c:114)
  ==30111==  by 0x818F94A: ose_shell_child (addrem.c:118)
  ==30111==  by 0x8105440: process_setup (targint.c:1508)
  ==30111==  by 0x4E2835A: start_thread (in /lib/libpthread-2.4.so)
  ==30111==  by 0x4D80C0D: clone (in /lib/libc-2.4.so)
  ```

  Now, all the functions that originate from the WRAP-OSE are filtered.

4.2.4 Limitation of Customized Valgrind

The following is a collection of limitation regarding customized Valgrind.

1. In theory the customized Valgrind should support all the platforms that the original Valgrind supports, but currently only x86/linux is tested
2. It is perfectly fine to use the customized Valgrind and its tools to analysis non-OSE cases, the results should not be affected. It is also fine to use “Check”/”UnCheck” suppression in non-OSE cases.
3. Recently, customized Valgrind only supports Memcheck and SGCheck .
4. Currently, the Customized Valgrind and its tools support the most recent version Valgrind 3.7.0. For the other Valgrind version, one may need to check the compatibility first. A very simple md5 check mechanism is provided. In the delivered package, there is a file called version.md5 that stores all the original md5 value of modified files. Please run "md5sum -c version.md5" before installation. If any file fails to pass md5 check, one may unfortunately need to check implementation details to find a solution to make it compatible. Please go to Appendix 1 for a list of modification in Valgrind source code.

4.3 OSE-WRAPPER

OSE-WRAPPER is an extra layer of OSE SFK that is created to help the Valgrind to find memory errors in OSE application. This section first gives a further analysis of the problem - why the original Valgrind Tools cannot detect memory errors in OSE applications and the solution to handle the problem in general, then two alternatives of implementing OSE-WRAPPER are presented in detail.

4.3.1 Further Problem analysis

Valgrind Tools like Memchek or SGcheck rely on detecting all the memory actions including memory reading, writing, allocation and de-allocation and changing the corresponding shadow values that records the status of each memory address to identify memory errors like memory leaks, out of boundary, accessing uninitialized memory units. When it comes to OSE cases, Valgrind can still automatically detect memory reading and writing but is blinded from most memory allocation/de-allocation, in that OSE uses customized resource allocation/de-allocation interface such as alloc(), heap_alloc(), free_buffer() that are not supported by Valgrind.

To un-blind Valgrind, OSE-WAPPER intercepts all the resource allocation/de-allocation system calls in BIOS level, then informs Valgrind about it properly. After this step, Valgrind is capable of detecting memory leak errors, but still cannot detect out of boundary and

accessing inaccessible memory units errors. That is due to the resource management mechanism of SFK, which first uses standard malloc() or mmap() to allocate a huge memory area from the host machine, from which the OSE customized resource allocation functions allocate buffers. Valgrind automatically detects and marks the memory area allocated by standard malloc() and mmap() as “accessible”, as a result, legalizing all the memory access inside this memory area. There are two alternatives to handle this problem.

1. OSE-WRAPPER Alternative 1 (A1) is marking the memory area that allocated by standard malloc() and mmap() back to “inaccessible”, but in this case, Valgrind misreports errors when any system call need to access meta data in this area. As a result, extra error suppressions need to be introduced to solve the problem.
2. OSE-WRAPPER Alternative 2 (A2) is creating local images for each signal/heap buffer by standard malloc, all the relevant system calls e.g heap_alloc(), alloc() are wrapped such that the return pointers of these system calls are actually the pointers of the corresponding local image. Namely, the OSE application, in this case, doesn’t read/write from the heap/signal buffers allocated by the kernel, but from the local images that are allocated by standard malloc() instead, therefore from Valgrind point of view, it is just a normal Linux application that uses stand resource allocation/de-allocation interfaces.

4.3.2 Classify the OSE resource allocation/de-allocation system call

To simplify the implementation and further discussion, all the system calls that relates to the signal buffer, are classified into 5 categories, alloc-like, free-buffer-like, send-like, receive-like, pointer-recover-like. And each category shares same features in a degree and can be handled in the same way in OSE-WRAPPER layer. There are other system calls that could not be classified into any category above which then need to be handled in a separated way. Please refer to Appendiex 2 for more details about how the system calls are sorted, this section only brief describes the common features that each category shares, which is as follow:

- Each alloc-like system call returns a pointer of a signal buffer.
- Despite of the difference of implementation and function purpose, all the send-like/receive-like system calls is a sense the same: send/receive a signal from one process to another.
- All the free-buffer-like system calls free the signal buffers, which have an argument that contains the address of the target signal buffer that needs to be freed.
- The pointer-recover-like system calls share the feature that all of them have an argument that contains a pointer to a signal buffer.

4.3.3 OSE-WRAPPER A1

4.3.3.1 Implementation Details of A1

The first solution is very straight-forward, by using Valgrind wrap function mechanism, all the relevant system calls are intercepted and instrumented by Valgrind user requests.

First, the heap and pool create system calls are intercepted, and the whole pool and heap data area are marked as “inaccessible” by Valgrind user request: VALGRIND_MAKE_ MEM_ NOACCESS.

Then, the alloc-like, receive-like and heap_share_alloc (heap_share_alloc is used to allocate share heap buffer, please go to section 2.2.5 for referencing the share heap buffer concept) system calls are instrumented by VALGRIND_MALLOC_LIKE client request; the send-like, free-buffer-like and free (free system call here denotes free share heap buffer) system calls are instrumented by VALGRIND_FREE_LIKE client request.

Meanwhile, all the private heap buffer allocation and de-allcoation system calls are instrumented by memory pool related client requests. Memory pool concept in Valgrind cannot be used to detect memory leaks, but it is idea for grouping memory blocks. These features are adorable for the private heap buffer, since in OSE, memory leak wouldn’t be not a issue for private heap buffer after garbage collection, meanwhile, in OSE, private heap buffers are grouped by the owner and freed by heap demon once the owner process dies, correspondingly, the create_process system call is intercept and instrument by VALGRIND_CREATE_POOL, and the private heap alloc and free system calls are intercepted and instrumented by VALGRIND_MEMPOOL_ALLOC and VALGRIND_ MEMPOOL_FREE respectively, such that all the private heap blocks are grouped by memory pool in term of their owners in Valgrind. In addition, a function called l_client_rem (when a process is killed, this function is then called by heap demon to free all the private heap buffers owned by that process) is intercepted and instrumented by VALGRIND_MEMPOOL_DESTORY to destroy the whole pool by Valgrind when the owner dies.

4.3.3.2 Error Detection

After the procedure above, Memcheck is capable of detecting all sorts of memory errors.

For leak errors, Memcheck records all the blocks informed by VALGRIND_MALLOC_LIKE client request and for each recorded block, Memcheck records all the corresponding pointers that point to it. A recorded block is kept in a tree structure maintained by Memcheck until VALGRIND_FREE_LIKE client request is called to free it. When the client program exits, Valgrind checks memory leaks errors by going through all the recorded blocks. If there is no pointer points to a recorded block, then a “definite lost” leak error is reported, otherwise Valgrind won’t report this block as memory leak unless a command-line option “--show-reachable=yes” is specified.

The mechanism of detecting out of boundary errors is similar to the red-zone concept. As is shown in Fig4.1 Valgrind first marks the whole heap or pool area as “inaccessible” (in Fig 4.1 the inaccessible area are marked as red), when the heap or pool buffers are allocated, Valgrind is informed and marks the buffers area as “accessible” (in Fig 4.1 the inaccessible area are marked as white), but the meta data and the endmarks of the buffers remain “inaccessible” which work as the redzone to check if the program access the memory area out of the boundary. Valgrind is capable of detect every memory reading/writing action and the corresponding memory address it accessed. Once Valgrind/Memcheck detects client program accessing an inaccessible memory unite, it report an invalid read/write error to indicate there is an out of boundary error.

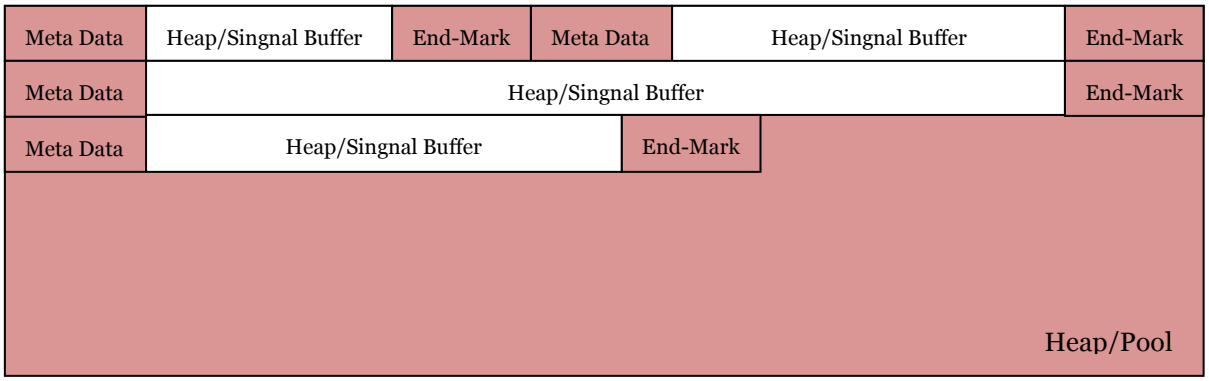

Fig 4.1 How Valgrind detect out of boundary errors (red: inaccessible, white: accessible)

For the other errors like accessing uninitialized memory unit and stack and global array overrun errors, Valgrind detects them as in normal applications context.

5.3.3.3 A Problem of OSE-WRAPPER A1

The biggest problem of this solution is that Memcheck reports false positive errors when the application calls certain system calls that in turn access meta data which are marked as “inaccessible”. The problem can be solved either by adding the following suppressions to the suppression file to suppress all the invalid memory access errors from all system calls, or by applying Alternative 2 – a totally different solution from Alternative 1.

4.3.4 OSE-WRAPPER A2

A22 is much more complicated than A1, the idea is based on creating a local image buffer on host for each OSE heap/signal buffer by standard malloc and then return the pointer of the local image buffers to the OSE application, such that from Valgrind aspect, it can detect errors just as in a normal non-OSE application.

4.3.4.1 Implementation Details of A2

Figure 4.2 presents the memory layout in A2.

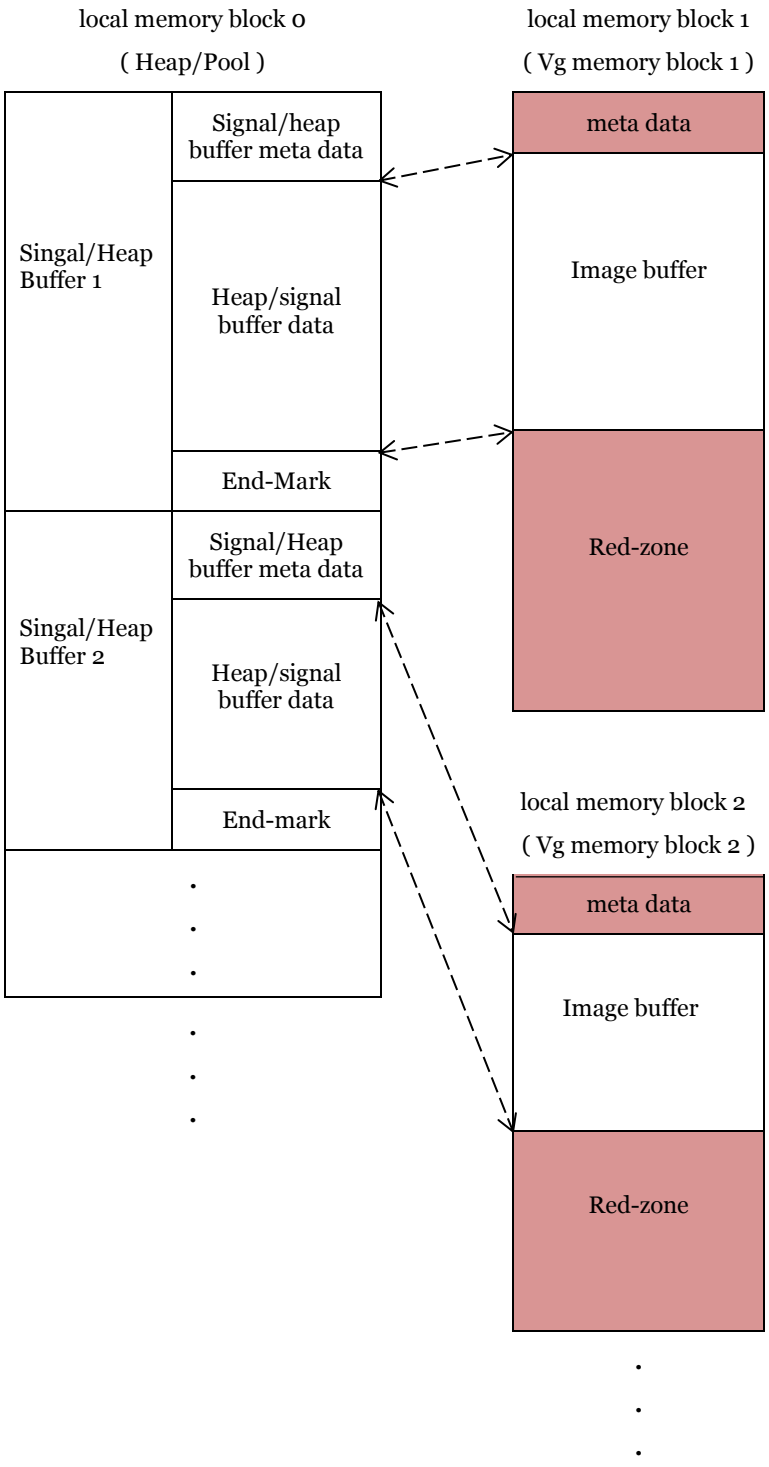

Figure 4.2 Memory layout of Alternative 2

In Figure 4.2, the local memory block is a memory block allocated by the host through standard *malloc()* or *mmap()* functions. Each signal/heap buffer allocated in that pool/heap are mapped to a corresponding local memory which is called Vg memory block (VB) in this thesis, Vg memory blocks are local memory blocks that are allocated by standard *malloc()* function and are recorded in a link list. A typical VB mainly contains following three parts:

1. Image buffer: a memory area of VB, which contains the copy of the corresponding

signal/heap buffer data, therefore the size of an image buffer is same as the size of corresponding signal/heap buffer.

2. Red-zone: a memory area of VB, which works as a buffer area in case that the image buffer needs to be extended (the image buffer may need to be extended, once OSE applications try to resize the corresponding heap/signal buffer). The initial size of Red-zone is the same as image buffer by default, but it shouldn't be either larger or smaller than the value defined by LARGEST_RED_ZONE macro and SMALLEST_RED_ZONE macro. Both of these two macros are defined in OSE-WRAPPER layer. The red-zone contains one byte of end-mark at the beginning address of red-zone area and the whole red-zone area is marked and remains as "unaccessible" unless the size of image buffer is changed.
3. The meta data of VB: a memory area of VB, which stores a data structure called _vg_buff, which contains a pointer to the next VB, a pointer that points to corresponding OSE signal/heap buffer, the size of image buffer, the owner of signal/heap buffer, a pointer that points to the image buffer address. The whole meta data area is marked as "unaccessible", meanwhile, all the functions in OSE_WRAPPER that need to access meta data area are added to the suppression file to suppress the invalid access errors, such that, OSE-WRAPPER functions could access any memory unit in meta data areas and Valgrind won't report any error.

Once alloc-like, receive-like or heap allocation system calls are called in OSE application, they are intercepted and wrapped. In the wrap function, the original system call runs first, then according to the result and the arguments of the system call, a corresponding VB is allocated and added to the head of the link list with the hope that the latest allocated signal/heap buffer will be freed first, after that the meta data of Vg memory block is initialized (for the receive-like system calls, an extra step is taken by the wrap function: copy the OSE signal/heap buffer data into image buffer in corresponding VB) finally instead of return the address of OSE heap/signal buffer, the start address of image buffer area is returned to the OSE application, such that the OSE application doesn't read/write OSE signal/heap buffer, but accesses the corresponding image buffer instead.

On the contrary, when free-like, send-like or heap free system calls are called by the OSE application, they are intercepted and wrapped as well. But instead of running the original system call first, the wrap function need convert the address of target free buffer from image buffer address back to OSE heap/signal buffer address (for the send-like system calls, an extra step is taken by the wrap function: copy the content of image buffer to OSE signal/heap buffer). After that, the wrap function calls original system call, frees the corresponding VB and removes it from the list.

As for pointer-recover-like system calls that have an argument that contains a pointer to signal buffer, OSE-WRAPPER intercepts and wraps these functions, converts the pointer from the address of image buffer in VB back to the address of corresponding OSE signal buffer, and calls the original system call in the end.

Like A1, when a private process is killed, A2 also frees all its private heap buffers' corresponding VB to simulate the OSE garbage collection functionality.

The link list and the VB that stored in the list are shared resources, which are protected by

POSIX mutex, which is possible in OSE SFK cases since each OSE SFK process is mapped to a corresponding POSIX pthread in Linux.

4.3.4.1 Conclusion of A2

In conclusion, A2 coverts OSE customized resource allocation/de-allocation interfaces into the standard ones without changing the behavior of the OSE application. In addition, through proper converting the addresses between the image buffer in VB and OSE heap/signal buffer, the OSE SFK doesn't realize the existence of OSE-WRAPPER layer and behaviors just as normal, while all the OSE customized resource allocation/de-allocation actions become transparent to Valgrind, thus Valgrind can detect all the memory errors of OSE application. Although, A2 is, in a sense, similar to the solution adopted by ENEA to apply Purify Plus to detect memory errors, there is a significant difference: A2 creates a separated layer of OSE to avoid modifying the source code of OSE core, while ENEA's solution is highly integrated in the OSE SFK source code.

4.3.5 Comparison between A1 and A2

The following conclude the differences and similarities of two alternatives

- Both alternatives relay on Valgrind wrap and client request mechanism to achieve BIOS level system call interception and instrumentation. As a result, the source code of original OSE SFK remain the same, the allocation/de-allocation actions in both kernel modules and load modules can be detected by Valgrind, and the instrumentation of OSE/OSEck kernel is avoided.
- Compared with OSE error checking mechanism which detects out of boundary errors by checking endmark overwritten. Both alternatives apply red zone like mechanism to detect out of boundary errors, Therefore the out of boundary error checking became more reliable. Besides, Valgrind could print the back trace of the errors which helps users to find the sources of the errors, which is impossible by pure OSE error checking mechanism.
- Both alternatives relay on customized Valgrind to suppress the errors that not relate to the OSE application. However, A1 tends to report false positive errors when application calls some interfaces which need to access the meta data in OSE heap or pools, therefore extra suppression type must be added to filter these kind of misreported errors and more test cases are needed to make A1 reliably work. While in A2, the OSE core could freely accesses the meta data without any error reported, but A2 consumes more memory foot-print since it maintains a copy for each heap/signal buffer.
- Due to OSE has garbage collection mechanism for private heap buffers, both alternatives only reports the memory leak errors for shared heap buffers.
- Due to A1 rudely marks the whole data area as "inaccessible", there is a huge number of errors needs to be recorded and suppressed by Valgrind, which could slow down the analyzing procedure. While in A2, the original code in OSE application is instrumented by extra memory allocating/deallocating and pointer converting actions, which could also in turn increase the execution time. In Appendix 3, there will be a

benchmark to check which alternative is faster.

4.4 Prototype Result

After the test that we have applied (For the testing information, please refer to Appendix 3 that gives the source code, the result and the analysis of each test case, while this section only simply conclude the results from Appendix 3), the prototype is proved to be capable of detecting all the memory errors as expected. As equivalent to Rational Purify, the prototype breaks the limitation of OSE error check mechanism, which can only detect one memory error – out of boundary under dedicated condition- the buffer's endmark is overwritten and cannot provide information to help the user to locate the source of the error. At Appendix 3, one can see that with acceptable speed penalty, the prototype not only can find all kinds of memory errors including memory leaks, out of heap/signal boundary errors, accessing uninitialized values, overrun of array, but also point out the source of the errors precisely. Especially the customized Valgrind with OSE-WRAPPER A2 solution, which has almost no speed penalty comparing with using original Valgrind, it is also faster, false positive error free and capable of provide precise extra error information compared to A1. When it comes to Rational Purify, it is even faster than the original Valgrind for analyzing the test program and it well supports GUI and runtime user interaction, however Purify also tends to report lots of OSE core errors and it fails to detect the array overrun error, which can be found by SGcheck, a Valgrind tool supported by the prototype implemented in this thesis.

5. Conclusion and Future works

This thesis first investigates the Rational PurifyPlus and Valgrind and analyzes the pros and con of both in detail. Then, different solutions of analyzing OSE/OSEck application through Valgrind tools are proposed and analyzed. In addition, a prototype that supports two Valgrind tool plug-in: Memcheck and SGcheck for detecting programming mistakes in application under the OSE SFK environments is successfully implemented and described. However, there are still limitations of current prototype, and the following future works are believed to be valuable:

1. The current prototype requires recompiling the OSE SFK with OSE-WRAPPER before using Valgrind tools to detect memory errors in OSE application. To remove the recompilation penalty, instead of creating an extra layer of OSE, it would be valuable to investigate solutions to integrate OSE-WRAPPER into Valgrind. The original Valgrind supports intercepting system calls like standard malloc(), calloc(), etc. It should be possible that applying the similar mechanism to intercept OSE system calls.
2. Currently, the modification on Valgrind is mainly on Valgrind core's suppression mechanism and Valgrind Tools still records all the errors which won't be reported to the users. To speeding up the debugging procedure, instead of letting Valgrind tools records all the errors and then suppressing them one by one, it could be a faster solution if Valgrind tools could avoid recording the errors that would be suppressed later. But in this case, extra modification is introduced to Valgrind Tool.
3. Although a couple of possible solutions are proposed in this paper, the current prototype doesn't support OSE Load module yet. It would be valuable to implement a stable solution to detect memory errors in OSE Load Modules.
4. The current prototype supports two Valgrind tools under the OSE SFK environments. But there are still works need to be done to make Valgrind Tools directly support OSE/OSEck targets.
5. Currently, limited Valgrind Tools are supported by the prototype: only Memcheck and SGcheck are customized for OSE application currently. There are some other ready-made standard Valgrind tools, e.g. Helgrind and DRD are capable of detecting thread errors, which could be quite useful for debugging OSE/OSEck application. Therefore, it would be valuable to investigate the solutions to customize more tools and even create new special Valgrind tools to support debugging and profiling OSE/OSEck application.

6. Reference

[1]. Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In Proceedings of PLDI 2007, San Diego, California, USA, June 2007.

[2]. Nicholas Nethercote and Julian Seward. How to Shadow Every Byte of Memory Used by a Program. In Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE 2007), San Diego, California, USA, June 2007.

[3]. M. Burrows, S. N. Freund, and J. L. Wiener. Run-time type checking for binary programs. In Proceedings of CC 2003, pages 90–105, Warsaw, Poland, April 2003.

[4]. Valgrind website: valgrind.org

[5]. Enea software AB website: www.enea.com

[6]. N. Nethercote. Dynamic Binary Analysis and Instrumentation. PhD thesis, University of Cambridge, United Kingdom, November 2004.

[7]. J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In Proceedings of the USENIX'05 Annual Technical Conference, Anaheim, California, USA, April 2005.

[8]. IBM Rational purify web page : http://www-01.ibm.com/software/awdtools/purify/

[9]. BoundsChecker Wikipedia page: http://en.wikipedia.org/wiki/BoundsChecker

[10]. Insure ++ web page http://www.parasoft.com/jsp/products/insure.jsp?itemId=63

[11]. ENEA products webpage: http://www.enea.com/software/products/rtos/ose/

[12]. Just-In-Time compilation concept: http://en.wikipedia.org/wiki/Just-in-time_compilation

[13]. Liedtke, Jochen "Towards Real Microkernels". Communications of the ACM 39 (9): 70–77, September 1996.

Appendix 1 – List of modified Valgrind source code

Modified files

1. *include/pub_tool_seqmatch.h*
2. *include/pub_tool_execontext.h*
3. *include/pub_tool_errormgr.h*
4. *memcheck/mc_error.c*
5. *coregrind/m_execontext.c*
6. *coregrind/m_errormgr.c*
7. *coregrind/m_seqmatch.c*

IN FILE: include/pub_tool_seqmatch.h

- **Add function declaration:**
  1. *Bool VG_(ose_generic_match) (*
     Bool matchAll,
     void patt, SizeT szbPatt, UWord nPatt, UWord ixPatt,*
     void input, SizeT szbInput, UWord nInput, UWord ixInput,*
     Bool using_app_symbol,
     *Bool (*pIsStar)(void*),*
     *Bool (*pIsQuery)(void*),*
     *Bool (*pattEQinp)(void*,void*)*
)
  2. *UWord VG_(ose_filter_match) (*
     Bool matchAll,
     void patt, SizeT szbPatt, UWord nPatt, UWord ixPatt,*
     void input, SizeT szbInput, UWord nInput, UWord ixInput,*
     Bool using_app_symbol,
     *Bool (*pIsStar)(void*),*
     *Bool (*pIsQuery)(void*),*
     *Bool (*pattEQinp)(void*,void*)*
)

IN FILE: include/pub_tool_execontext.h

- **Add function declaration:**
  1. *extern int* VG_(get_ExeContext_n_ips_addr)(ExeContext* e) ;*

IN FILE: include/pub_tool _errormgr.h

- **Add function declaration:**
  1. *void VG_(ose_filter_ExeContext)(ExeContext *e);*

IN FILE: include/mc_error.c

- **Modify function:**
  1. *static void mc_pp_AddrInfo (Addr a, AddrInfo* ai, Bool maybe_gcc)*

IN FILE: include/mc_error.c

- **Modify function:**
  1. *static void load_one_suppressions_file (Char* filename);*
  2. *void VG_(load_suppressions) (void)*
  3. *void VG_(maybe_record_error) (ThreadId tid, ErrorKind ekind,*
     Addr a, Char s, void* extra);*
  4. *Bool VG_(unique_error) (ThreadId tid, ErrorKind ekind, Addr a, Char* s,*
     void extra, ExeContext* where, Bool print_error,*
     Bool allow_db_attach, Bool count_error);

- **Add function:**
  1. *static Supp* is_ose_krn_suppressible_error (Error* err);*
  2. *static Bool ose_recognised_suppression (Char* name, Supp* su);*
  3. *static Supp* get_ose_filter_supp(void);*
  4. *static void ose_filter_erro(Error* err);*
  5. *static Bool ose_supp_matches_callers(Error* err, Supp* su);*
  6. *static VG_(ose_filter_ExeContext)(ExeContext *e);*

- **Add macro:**
  1. *#define VG_MAX_SUPP_CALLERS_OSE 200*
  2. *#define CHECKFUNS 100*
  3. *#define UNCHECKFUNS 101*
  4. *#define FILTERFUNS 102*

- **Add global variable:**
  1. *static Supp *ose_suppressions=NULL;*
  2. *static Supp *ose_filter_suppressions=NULL;*

3. *static UInt ose_mode=2;*

IN FILE: coregrind/m_seqmatch.c

- **Add functions:**

1. *Bool VG_(ose_generic_match) (*
   Bool matchAll,
   void patt, SizeT szbPatt, UWord nPatt, UWord ixPatt,*
   void input, SizeT szbInput, UWord nInput, UWord ixInput,*
   Bool using_app_symbol,
   *Bool (*pIsStar)(void*),*
   *Bool (*pIsQuery)(void*),*
   *Bool (*pattEQinp)(void*,void*)*
)
2. *UWord VG_(ose_filter_match) (*
   Bool matchAll,
   void patt, SizeT szbPatt, UWord nPatt, UWord ixPatt,*
   void input, SizeT szbInput, UWord nInput, UWord ixInput,*
   Bool using_app_symbol,
   *Bool (*pIsStar)(void*),*
   *Bool (*pIsQuery)(void*),*
   *Bool (*pattEQinp)(void*,void*)*
)

Appendix 2 – Instrumented system calls

All the following system calls are BIOS-level intercepted and properly instrumented in OSE-WRAPPER layer.

- The following definitions of syntax OSE_EXEC_FC_<system call> define the system call function codes used by OSE exec. Meanwhile classifying these system calls in terms of alloc-like, send-like, receive-like, free-buffer-like and pointer-recover-like categories. The purpose of this classification work is that each category shares the same features in a degree, such that can be handled in the same way in OSE-WRAPPER layer.
    - **Alloc-like system call number:**
      All the alloc-like system calls return a pointer of the signal buffer

      #define OSE_EXEC_FC_alloc 2
      #define OSE_EXEC_FC_get_bid_list 19
      #define OSE_EXEC_FC_get_cpu 20
      #define OSE_EXEC_FC_get_env 21
      #define OSE_EXEC_FC_get_env_list 22
      #define OSE_EXEC_FC_get_pcb 25
      #define OSE_EXEC_FC_get_pid_list 26
      #define OSE_EXEC_FC_get_sem 29
      #define OSE_EXEC_FC_get_signal 30
      #define OSE_EXEC_FC_alloc_nil 73

    - **Send-like system call number**
      Despite of the difference of implementation and function purpose, all the send-like system calls is a sense the same: send a signal from one process to another

      #define OSE_EXEC_FC_attach 4
      #define OSE_EXEC_FC_hunt 34
      #define OSE_EXEC_FC_hunt_from 35
      #define OSE_EXEC_FC_send 45
      #define OSE_EXEC_FC_send_w_s 47
      #define OSE_EXEC_FC_ose_fsend 75
      #define OSE_EXEC_FC_zzattach 85
      #define OSE_EXEC_FC_zzhunt 92
      #define OSE_EXEC_FC_zzsend 98

#define OSE_EXEC_FC_zzsend_w_s 99
#define OSE_EXEC_FC_zzhunt_from 110

- **Free-buffer-like system call number**
  Free-buffer-like system calls free the signal buffers, the difference between two free buffer system calls above is that OSE_EXEC_FC_zzfree_buf works in debug mode.

  #define OSE_EXEC_FC_free_buf 17
  #define OSE_EXEC_FC_zzfree_buf 91

- **Receive-like system call number**
  Despite of the difference of implementation and function purpose, all the receive-like system calls is a sense the same: receive a signal from one process to another

  #define OSE_EXEC_FC_receive 40
  #define OSE_EXEC_FC_receive_from 41
  #define OSE_EXEC_FC_receive_w_tmo 42
  #define OSE_EXEC_FC_receive_with 84
  #define OSE_EXEC_FC_zzreceive 95
  #define OSE_EXEC_FC_zzreceive_from 96
  #define OSE_EXEC_FC_zzreceive_w_tmo 97
  #define OSE_EXEC_FC_zzreceive_with 111

- **Pointer-recover-like system call number**
  The pointer-recover-like system calls share the feature that all of them have an argument that contains a pointer to signal buffer.

  #define OSE_EXEC_FC_addressee 1
  #define OSE_EXEC_FC_restore 43
  #define OSE_EXEC_FC_sender 46
  #define OSE_EXEC_FC_sigsize 54
  #define OSE_EXEC_FC_zzsender 112
  #define OSE_EXEC_FC_zzsigsize 114

The get_sysconf system call itself does not belong to any category above, however it contains sub-system calls, and some of them can be classified, while some of them cannot and in turn need special treatments. The function codes of get_sysconf system call and its debug mode equivalent are as following:

#define OSE_EXEC_FC_get_sysconf 74
#define OSE_EXEC_FC_zzget_sysconf 109

- The following defines HEAP bios function codes:

```
#define FC_FORCED_FREE          16
#define FC_PLUG_ALLOC           25
#define FC_PLUG_FREE            26
#define FC_PLUG_RESIZE          27
```

Appendix 3 – Test cases and test result

This appendix presents and explains the source code of the test cases that have been used in this thesis. In addition, this appendix also compares and analyses the output of OSE kernel dump, implemented prototype (Customized Valgrind together with OSE-WRAPPER A1/A2) and the Rational Purify in terms of detecting memory errors for each test case.

A3.1 Test Case 1 – Limitation of OSE error check

The test case 1 is created to show the limitations of pure OSE error check in terms of detecting out of boundary errors, by comparing the output of OSE kernel dump, Rational Purify and the prototype implemented in this thesis.

OSE can detect out of boundary errors only when the buffers' endmarks are modified. Once the error is founded, OSE kernel reports a fatal error and exits the program. Therefore it is necessary to build a separate test for OSE error check, since if there are errors that occur after the endmark overwriting, Rational Purify and Valgrind Tools cannot detect them either, meanwhile Purify and Memcheck do the leak check after the program exit, namely if the OSE SFK aborts in the middle of execution, Purify and Memcheck cannot report leak errors precisely.

A3.1.1 Source Code of Test Case

The source code of test case *test1.c* is as following, from which we can see 2 out of boundary errors in line 57 and line 60 respectively. The first error is accessing a out of boundary memory unit without overwriting the buffer's endmark, while the second one is accessing a out of boundary memory unit by modifying the signal buffer's endmark.

test1.c:

```
27  #include "ose.h"
28  #include "efs.h"
29  #include "stdio.h"
30  #include "string.h"
31  #include "malloc.h"
32
33  union SIGNAL
34  {
35      SIGSELECT signo;
36  };
```

```
37
38  #define PRINT_SIG 1000
39  struct print_sig
40  {
41          SIGSELECT signo;
42          char text [1]; /* Variable size. */
43  };
44  const char test1_synopsis [] = "hello";
45  const char test1_descr [] = "print hello world!";
46
47   int test1_main(int argc, char *argv[]) {
48          struct print_sig *prsig;
49
50          /* get the PID of current process */
51          PROCESS test1_pid = current_process ();
52
53          /* Allocate a signal buffer with size 10 bytes */
54          prsig = (struct print_sig *)alloc(10, PRINT_SIG);
55
56          /* Error: access a memory unit out of boundary */
57          ((char*) prsig)[14]='a';
58
59          /* Error: overwrite endmark of the signal buffer */
60          ((char*) prsig)[10]='a';
61
62           /*send the signal to current processor itself,
63              the send system call is called here, such that
64             the OSE kernel will perform endmark overwritten check*/
65           send ((union SIGNAL **)&prsig, test1_pid);
66
67          /* Return success from my command. */
68          return 0;
69  }
```

A3.1.2 Output from OSE SFK

The following message is extracted from OSE kernel dump, which indicates that during *send()* system call, OSE kernel did the error check and found the endmark of signal buffer whose address is 0xf6929040 is overwritten. But the message doesn't indicate the source of the error, meanwhile, OSE kernel only successfully detects one of the out of boundary errors.

SEH: Kernel detected FATAL error 0x802d00a5 extra 0xf6929040
SEH: System call: send
SEH: Error: Buffer endmark for signal @ 0xf6929040 overwritten

A3.1.3 Output of Customized Valgrind/Memcheck with OSE-WRAPPER A1

The following is the error message output from customized Valgrind/Memcheck with OSE-WRAPPER A1, from which we can see that Valgrind not only finds all the out of boundary errors, but also indicates the source of the errors: test1.c: 57 and test1.c:60. The only drawback is about the extra information of invalid write errors, Valgrind either fails to display which signal buffer's boundary is overran or just gives the wrong information. Fortunately, this extra information is usually not that important to either identify or locate the errors.

==4289== Thread 44:
==4289== Invalid write of size 1
==4289== at 0x806417F: test1_main (test1.c:57)
==4289== by 0x818F8EA: ose_shell_child (addrem.c:118)
==4289== by 0x81053E0: process_setup (targint.c:1508)
==4289== by 0x4E2835A: start_thread (in /lib/libpthread-2.4.so)
==4289== by 0x4D80C0D: clone (in /lib/libc-2.4.so)
==4289== Address 0x8f5b06e is 237,638 bytes inside a block of size 4,194,304 alloc'd
==4289== at 0x4C9B12E: malloc (vg_replace_malloc.c:263)
==4289== by 0x805CC55: krnconf_config_syspool (krncon.c:1070)
==4289== by 0x8064553: krnconf_config_syspool (wrap_ose.c:1334)
==4289== by 0x805CED6: krnconf_read_config (krncon.c:1143)
==4289== by 0x805D5AA: zzose_con_1 (krncon.c:1766)
==4289== by 0x80FFF0C: exec_start (exec_start.c:387)
==4289== by 0x810014D: zzstart_OSE (exec_start.c:584)
==4289== by 0x805B3DA: main (osemain.c:605)
==4289==
==4289== Invalid write of size 1
==4289== at 0x8064188: test1_main (test1.c:60)
==4289== by 0x818F8EA: ose_shell_child (addrem.c:118)
==4289== by 0x81053E0: process_setup (targint.c:1508)
==4289== by 0x4E2835A: start_thread (in /lib/libpthread-2.4.so)
==4289== by 0x4D80C0D: clone (in /lib/libc-2.4.so)
==4289== Address 0x8f5b06a is 237,634 bytes inside a block of size 4,194,304 alloc'd
==4289== at 0x4C9B12E: malloc (vg_replace_malloc.c:263)
==4289== by 0x805CC55: krnconf_config_syspool (krncon.c:1070)
==4289== by 0x8064553: krnconf_config_syspool (wrap_ose.c:1334)
==4289== by 0x805CED6: krnconf_read_config (krncon.c:1143)
==4289== by 0x805D5AA: zzose_con_1 (krncon.c:1766)
==4289== by 0x80FFF0C: exec_start (exec_start.c:387)
==4289== by 0x810014D: zzstart_OSE (exec_start.c:584)
==4289== by 0x805B3DA: main (osemain.c:605)
==4289==
==4289== HEAP SUMMARY:
==4289== in use at exit: 4,262,940 bytes in 184 blocks

==4289== total heap usage: 1,921 allocs, 1,664 frees, 4,545,055 bytes allocated
==4289==
==4289== LEAK SUMMARY:
==4289== definitely lost: 0 bytes in 0 blocks
==4289== indirectly lost: 0 bytes in 0 blocks
==4289== possibly lost: 0 bytes in 0 blocks
==4289== still reachable: 0 bytes in 0 blocks
==4289== suppressed: 95,493 bytes in 246 blocks
==4289==
==4289== For counts of detected and suppressed errors, rerun with: -v
==4289== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 1417861 from 16)

A3.1.4 Output of Customized Valgrind/Memcheck with OSE-WRAPPER A2

The following is the error message output from customized Valgrind/Memcheck with OSE-WRAPPER A2, from which we can see that Valgrind not only finds all the out of boundary errors and indicates the source of the errors: test1.c: 57 and test1.c:60, but also successfully displays the information that the program accesses the memory units that outside of the boundary of the signal buffer allocated at *test1_main (test1.c:54)*.

==18877== Thread 44:
==18877== Invalid write of size 1
==18877== at 0x806417F: test1_main (test1.c:57)
==18877== by 0x8191A9A: ose_shell_child (addrem.c:118)
==18877== by 0x8107588: process_setup (targint.c:1508)
==18877== by 0x4E2535A: start_thread (in /lib/libpthread-2.4.so)
==18877== by 0x4D85C0D: clone (in /lib/libc-2.4.so)
==18877== Address 0x72ccace is 46 bytes inside a block of size 53 alloc'd
==18877== at 0x81A6D87: alloc (softkrnflib.c:104)
==18877== by 0x8064175: test1_main (test1.c:54)
==18877== by 0x8191A9A: ose_shell_child (addrem.c:118)
==18877== by 0x8107588: process_setup (targint.c:1508)
==18877==
==18877== Invalid write of size 1
==18877== at 0x8064188: test1_main (test1.c:60)
==18877== by 0x8191A9A: ose_shell_child (addrem.c:118)
==18877== by 0x8107588: process_setup (targint.c:1508)
==18877== by 0x4E2535A: start_thread (in /lib/libpthread-2.4.so)
==18877== by 0x4D85C0D: clone (in /lib/libc-2.4.so)
==18877== Address 0x72ccaca is 42 bytes inside a block of size 53 alloc'd
==18877== at 0x81A6D87: alloc (softkrnflib.c:104)
==18877== by 0x8064175: test1_main (test1.c:54)
==18877== by 0x8191A9A: ose_shell_child (addrem.c:118)
==18877== by 0x8107588: process_setup (targint.c:1508)
==18877==

==18877== HEAP SUMMARY:
==18877== in use at exit: 4,341,186 bytes in 408 blocks
==18877== total heap usage: 2,151 allocs, 1,743 frees, 4,712,760 bytes allocated
==18877==
==18877== LEAK SUMMARY:
==18877== definitely lost: 0 bytes in 0 blocks
==18877== indirectly lost: 0 bytes in 0 blocks
==18877== possibly lost: 0 bytes in 0 blocks
==18877== still reachable: 0 bytes in 0 blocks
==18877== suppressed: 4,341,186 bytes in 408 blocks
==18877==
==18877== For counts of detected and suppressed errors, rerun with: -v
==18877== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 321518 from 0)

A3.1.5 Output of Rational Purify

The following is the error message extracted from the output from Rational Purify. From which we can see that Purify finds all the out of boundary errors, but it also finds other 3000 errors from OSE core, although Purify support run time error suppressing, manually suppressing 3000 errors still could be fairly annoying.

ABW: Array bounds write:
** This is occurring while in thread 11673:*
test1_main [test1.c:57]
ose_shell_child [addrem.c:118]
process_setup [targint.c:1508]
start_thread [libpthread.so.0]
clone [libc.so.6]
** Writing 1 byte to 0x90ccace in the heap.*
** Address 0x90ccace is 4 bytes past end of a malloc'd block at 0x90ccac0 of 11 bytes.*
** This block was allocated from thread 11673:*
malloc [rtlib.o]
zzpure_alloc [targint.c:3717]
zzsystem_callh [targint.c:4221]
zzbiosCall [bios_c.c:193]
biosCall [biosflib.c:208]
alloc [softkrnflib.c:104]
test1_main [test1.c:54]
ose_shell_child [addrem.c:118]
process_setup [targint.c:1508]
start_thread [libpthread.so.0]
clone [libc.so.6]

***** Purify instrumented obj/rtose_debug/rtose (pid 11613) *****

ABW: Array bounds write:

** This is occurring while in thread 11673:*
test1_main [test1.c:60]
ose_shell_child [addrem.c:118]
process_setup [targint.c:1508]
start_thread [libpthread.so.0]
clone [libc.so.6]
** Writing 1 byte to 0x90ccaca in the heap.*
** Address 0x90ccaca is 10 bytes into a malloc'd block at 0x90ccac0 of 11 bytes.*
** This block was allocated from thread 11673:*
malloc [rtlib.o]
zzpure_alloc [targint.c:3717]
zzsystem_callh [targint.c:4221]
zzbiosCall [bios_c.c:193]
biosCall [biosflib.c:208]
alloc [softkrnflib.c:104]
test1_main [test1.c:54]
ose_shell_child [addrem.c:118]
process_setup [targint.c:1508]
start_thread [libpthread.so.0]
clone [libc.so.6]

A3.1.6 Conclusion of Test Case 1

As is shown from the results from test case 1, we can see that OSE error check mechanism could only detect out of boundary errors when the endmarks of the buffers are overwritten. Besides, even if the error is detected, OSE kernel cannot help the user to find the source of the error. While Rational Purify and the Customized Valgrind with OSE-WRAPPER A1/A2 could find all kinds of out of boundary errors and indicates source of the errors. The Customized Valgrind with OSE-WRAPPER A2 and Rational Purify can even precisely provide the address of the buffer whose boundary is overran. However Purify lacks the mechanism to filter OSE core errors, consequently it reports more than 3000 errors in the end.

A3.2 Test Case 2 – Prototype functionality test

The test case 2 is created to check if the prototype could find the memory errors that the original Memcheck and SGcheck could find. Meanwhile this section compares and analyses the output of OSE kernel dump, implemented prototype (Customized Valgrind together with OSE-WRAPPER A1/A2) and the Rational Purify in terms of detecting memory errors in test case 2

A3.2.1 Source Code of Test Case

The source code of test case 2 is as following. From which we can see that there are 6 memory errors. To be specific, in “world” process, there are three memory errors: (a) Array overrun error at line 58 which should be detected by SGcheck; (b) Out of signal buffer boundary error at line 72 which should be detected by Memcheck; (c) Fail to free signal buffer, memory leak

at line 84 which should be detect by Memcheck. And in"test2_main" process, there are also three memory errors: (a) Fail to free shared heap buffer, memory leak at line 111 which should be detected by Memcheck. (b) Using uninitialized value error at line 112 which should be detected by Memcheck. (c) Out of heap buffer boundary error at line 113 which should be detected by Memcheck.

test2.c

```
36      #include "ose.h"
37      #include "efs.h"
38      #include "stdio.h"
39      #include "string.h"
40      #include "malloc.h"
41      union SIGNAL
42      {
43            SIGSELECT signo;
44     };
45
46      #define PRINT_SIG 1000
47     struct print_sig
48      {
49            SIGSELECT signo;
50           char text[1]; /* Variable size. */
51      };
52
53      static const char msg [] = "world!\n";
54
55      static OS_PROCESS (world) {
56            static const SIGSELECT sel_any [] = { 0 };
57             union SIGNAL *sig;
58
59              int i, a[10];                  // both are auto vars
60             for (i = 0; i <= 10; i++)  //ERROR: array overrun
61                   a[i] = 42;
62
63            /* Receive first signal in signal queue. */
64             sig = receive(sel_any);
65
66            /* Check signal number. */
67            switch (sig->signo)
68            {
69                     /* Act on the PRINT_SIG signal. */
70                   case PRINT_SIG :
71                   {
72                         struct print_sig *prsig = (struct print_sig *)sig;
73                         // Error: out of signal buffer boundary
```

```
74                      ((char*) prsig)[sizeof(struct print_sig) + strlen(msg)+8]='a';
75                      fputs (prsig->text, stdout);
76                      fflush (stdout);
77                      break;
78                  }
79
80                  /* Call error() for all unexpected signals. */
81                   default:
82                       error2 (0xFFFFFFFE, (OSERRCODE)sig);
83                      break;
84              }
85              // Error: fail to free signal buffer, memory leak error.
86              //      free_buf (&sig);
87              /* Terminate self (open files are NOT flushed and closed). */
88              kill_proc (current_process());
89      }
90
91      const char test2_synopsis [] = "hello";
92      const char test2_descr [] = "print hello world!";
93
94      int test2_main (int argc, char *argv[]) {
95
96              PROCESS hello_pid = current_process ();
97              PROCESS world_pid;
98              static const SIGSELECT sel_attach[] = { 1, OS_ATTACH_SIG };
99              union SIGNAL *sig;
100             struct print_sig *prsig;
101
102             /* Create world process with same type and priority. */
103             world_pid = create_process(get_ptype(hello_pid), /* Process type. */
104                                                     "world",          /* Name. */
105                                                         world,          /* Entrypoint. */
106                                                         1000,           /* Stacksize. */
107                                                       get_pri(hello_pid),  /* Priority. */
108                                                   0, 0, NULL, 0, 0);   /* Uninteresting. */
109
110     // Error:  ph is not freed but no error report since it is a private heap alloc
111     //  by contrast, sh is not freed but report as a leak error since it is a
112     //  shared heap buffer, namely it won't be taken care by garbage   collection
113             int *ph=malloc (5*sizeof (int));
114             int *sh=heap_alloc_shared (5*sizeof (int),NULL,0);
115             if ( sig ! =2)                          //Error: use uninitailized value
116              ph [5*sizeof (int)+4]=1;               //Error: out of heap buffer boundary
117
118             /* Supervise world process (attach default signal to it). */
```

```
119             attach (NULL, world_pid);
120
121             /* Make world process inherit my stdio etc. */
122             efs_clone (world_pid);
123
124             /* Allocate, initialize and send signal with text to print. */
125             prsig = (struct print_sig *)alloc(sizeof(struct print_sig) + strlen(msg),
126                         PRINT_SIG);
127              strcpy (prsig->text, msg);
128                   send ((union SIGNAL **) & prsig, world_pid);
129
130             /* Print first part of message. */
131             printf ("Hello "); fflush(stdout);
132
133             /* Start world process. */
134             start (world_pid);
135
136             /* Wait for world process to terminate. */
137             sig = receive (sel_attach);
138              free_buf (&sig);
139
140             /* Return success from my command. */
141              return 0;
142     }
```

A3.2.2 Output from OSE SFK

OSE SFK doesn't detect any error in this case.

A3.2.3 Output of Customized Valgrind Tools with OSE-WRAPPER A1

- **Output from Memcheck:**
  The following is the error message output from customized Valgrind/Mecheck with OSE-WRAPPER A1. From which we can see that except the array overrun error, Memcheck detects all the errors and successfully indicates the source of the errors. However, when it comes to extra information of invalid write errors, Valgrind either fails to display which signal buffer's boundary is overran or just gives the wrong information. Fortunately, this extra information is usually not that important to either identify or locate the errors. Besides, Memcheck misreport one leak error, which is clearly not from user application and could be suppressed latter by adding a customized suppression file to avoid the same misreporting happening again.

```
==30101== Thread 44:
==30101== Conditional jump or move depends on uninitialised value(s)
==30101==    at 0x8064349: test2_main (test2.c:115)
==30101==    by 0x818F94A: ose_shell_child (addrem.c:118)
```

==30101== by 0x8105440: process_setup (targint.c:1508)
==30101== by 0x4E2835A: start_thread (in /lib/libpthread-2.4.so)
==30101== by 0x4D80C0D: clone (in /lib/libc-2.4.so)
==30101==
==30101== Invalid write of size 4
==30101== at 0x8064351: test2_main (test2.c:116)
==30101== by 0x818F94A: ose_shell_child (addrem.c:118)
==301n01== by 0x8105440: process_setup (targint.c:1508)
==30101== by 0x4E2835A: start_thread (in /lib/libpthread-2.4.so)
==30101== by 0x4D80C0D: clone (in /lib/libc-2.4.so)
==30101== Address 0x2000f860 is not stack'd, malloc'd or (recently) free'd
==30101==
==30101== Thread 45:
==30101== Invalid write of size 1
==30101== at 0x806421F: world (test2.c:74)
==30101== by 0x8105440: process_setup (targint.c:1508)
==30101== by 0x4E2835A: start_thread (in /lib/libpthread-2.4.so)
==30101== by 0x4D80C0D: clone (in /lib/libc-2.4.so)
==30101== Address 0x8f5baf7 is 8 bytes after a block of size 15 free'd
==30101== at 0x806731E: zzsystem_callh (wrap_ose.c:1703)
==30101== by 0x8153776: zzbiosCall (bios_c.c:193)
==30101== by 0x8068184: zzbiosCall (wrap_ose.c:1839)
==30101== by 0x81A4AD2: biosCall (biosflib.c:208)
==30101== by 0x81A5DE9: zzsend (softkrnflib.c:781)
==30101== by 0x80643DE: test2_main (test2.c:128)
==30101== by 0x818F94A: ose_shell_child (addrem.c:118)
==30101== by 0x8105440: process_setup (targint.c:1508)
==30101== by 0x4E2835A: start_thread (in /lib/libpthread-2.4.so)
==30101== by 0x4D80C0D: clone (in /lib/libc-2.4.so)
==30101==
==30101==
==30101== HEAP SUMMARY:
==30101== in use at exit: 4,263,169 bytes in 188 blocks
==30101== total heap usage: 2,193 allocs, 1,906 frees, 4,554,669 bytes allocated
==30101==
==30101== Thread 1:
==30101== 0 bytes in 1 blocks are definitely lost in loss record 1 of 183
==30101== at 0x818A2EF: heap_alloc (heapapi.c:486)
==30101== by 0x818A993: heap_alloc_shared (heapapi.c:769)
==30101== by 0x806433F: test2_main (test2.c:114)
==30101== by 0x818F94A: ose_shell_child (addrem.c:118)
==30101== by 0x8105440: process_setup (targint.c:1508)
==30101== by 0x4E2835A: start_thread (in /lib/libpthread-2.4.so)
==30101== by 0x4D80C0D: clone (in /lib/libc-2.4.so)
==30101==

==30101== 15 bytes in 1 blocks are definitely lost in loss record 20 of 183
==30101== at 0x81A5D2A: zzreceive (softkrnflib.c:758)
==30101== by 0x80641F4: world (test2.c:64)
==30101== by 0x8105440: process_setup (targint.c:1508)
==30101== by 0x4E2835A: start_thread (in /lib/libpthread-2.4.so)
==30101== by 0x4D80C0D: clone (in /lib/libc-2.4.so)
==30101==
==30101== 44 bytes in 1 blocks are definitely lost in loss record 90 of 183
==30101== at 0x81A4C37: alloc (softkrnflib.c:104)
==30101== by 0x8189181: ose_get_ppdata (pflib.c:1134)
==30101== by 0x8193510: efs_get_flib_vars (flib.c:507)
==30101== by 0x81A446A: efs_stdout (stdout.c:35)
==30101== by 0x8064226: world (test2.c:75)
==30101== by 0x8105440: process_setup (targint.c:1508)
==30101== by 0x4E2835A: start_thread (in /lib/libpthread-2.4.so)
==30101== by 0x4D80C0D: clone (in /lib/libc-2.4.so)
==30101==
==30101== LEAK SUMMARY:
==30101== definitely lost: 59 bytes in 3 blocks
==30101== indirectly lost: 0 bytes in 0 blocks
==30101== possibly lost: 0 bytes in 0 blocks
==30101== still reachable: 0 bytes in 0 blocks
==30101== suppressed: 95,687 bytes in 248 blocks
==30101==
==30101== For counts of detected and suppressed errors, rerun with: -v
==30101== Use --track-origins=yes to see where uninitialised values come from
==30101== ERROR SUMMARY: 6 errors from 6 contexts (suppressed: 588143 from 192)

- **Output from SGcheck:**
  The following is the error message output from customized Valgrind/SGcheck with OSE-WRAPPER A1. From which we can see that SGcheck detects the out of array overrun errors and points out exactly the source of the error.

==28951== Thread 45:
==28951== Invalid write of size 4
==28951== at 0x80641C7: world (test2.c:61)
==28951== by 0x8105440: process_setup (targint.c:1508)
==28951== by 0x4DC235A: start_thread (in /lib/libpthread-2.4.so)
==28951== by 0x4D2AC0D: clone (in /lib/libc-2.4.so)
==28951== Address 0x2a30f424 expected as actual:
==28951== Expected: stack array "a" of size 40 in this frame
==28951== Actual: unknown
==28951== Actual: is 0 after Expected
==28951==
==28951== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 42 from 0)

A3.2.4 Output of Customized Valgrind Tools with OSE-WRAPPER A2

- **Output from Memcheck:**
  The following is the error message output from customized Valgrind/Mecheck with OSE-WRAPPER A2. From which we can see that except the out of array overrun error, Memcheck not only successfully detects and locates the source of the errors, but also precisely reports the extra information for invalid write errors.

```
==23949== Thread 44:
==23949== Conditional jump or move depends on uninitialised value(s)
==23949==    at 0x8064349: test2_main (test2.c:115)
==23949==    by 0x8191A9A: ose_shell_child (addrem.c:118)
==23949==    by 0x810758C: process_setup (targint.c:1508)
==23949==    by 0x4E2535A: start_thread (in /lib/libpthread-2.4.so)
==23949==    by 0x4D85C0D: clone (in /lib/libc-2.4.so)
==23949==
==23949== Invalid write of size 4
==23949==    at 0x8064351: test2_main (test2.c:116)
==23949==    by 0x8191A9A: ose_shell_child (addrem.c:118)
==23949==    by 0x810758C: process_setup (targint.c:1508)
==23949==    by 0x4E2535A: start_thread (in /lib/libpthread-2.4.so)
==23949==    by 0x4D85C0D: clone (in /lib/libc-2.4.so)
==23949==  Address 0x72cbe00 is 0 bytes inside a block of size 73 alloc'd
==23949==    at 0x818C43F: heap_alloc (heapapi.c:486)
==23949==    by 0x818CAE3: heap_alloc_shared (heapapi.c:769)
==23949==    by 0x806433F: test2_main (test2.c:114)
==23949==    by 0x8191A9A: ose_shell_child (addrem.c:118)
==23949==    by 0x810758C: process_setup (targint.c:1508)
==23949==    by 0x4E2535A: start_thread (in /lib/libpthread-2.4.so)
==23949==
==23949== Thread 45:
==23949== Invalid write of size 1
==23949==    at 0x806421F: world (test2.c:74)
==23949==    by 0x810758C: process_setup (targint.c:1508)
==23949==    by 0x4E2535A: start_thread (in /lib/libpthread-2.4.so)
==23949==    by 0x4D85C0D: clone (in /lib/libc-2.4.so)
==23949==  Address 0x72ce607 is 55 bytes inside a block of size 63 alloc'd
==23949==    at 0x81A7E7A: zzreceive (softkrnflib.c:758)
==23949==    by 0x80641F4: world (test2.c:64)
==23949==    by 0x810758C: process_setup (targint.c:1508)
==23949==
==23949== HEAP SUMMARY:
==23949==     in use at exit: 4,341,791 bytes in 414 blocks
==23949==   total heap usage: 2,402 allocs, 1,988 frees, 4,737,216 bytes allocated
==23949==
==23949== Thread 1:
```

```
==23949== 63 bytes in 1 blocks are definitely lost in loss record 37 of 203
==23949==    at 0x81A7E7A: zzreceive (softkrnflib.c:758)
==23949==    by 0x80641F4: world (test2.c:64)
==23949==    by 0x810758C: process_setup (targint.c:1508)
==23949==
==23949== 73 bytes in 1 blocks are definitely lost in loss record 50 of 203
==23949==    at 0x818C43F: heap_alloc (heapapi.c:486)
==23949==    by 0x818CAE3: heap_alloc_shared (heapapi.c:769)
==23949==    by 0x806433F: test2_main (test2.c:114)
==23949==    by 0x8191A9A: ose_shell_child (addrem.c:118)
==23949==    by 0x810758C: process_setup (targint.c:1508)
==23949==    by 0x4E2535A: start_thread (in /lib/libpthread-2.4.so)
==23949==
==23949== LEAK SUMMARY:
==23949==    definitely lost: 136 bytes in 2 blocks
==23949==    indirectly lost: 0 bytes in 0 blocks
==23949==      possibly lost: 0 bytes in 0 blocks
==23949==    still reachable: 0 bytes in 0 blocks
==23949==         suppressed: 4,341,655 bytes in 412 blocks
==23949==
==23949== For counts of detected and suppressed errors, rerun with: -v
==23949== Use --track-origins=yes to see where uninitialised values come from
==23949== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 268325 from 0)
```

- **Output from SGcheck:**
  The following is the error message output from customized Valgrind/SGcheck with OSE-WRAPPER A2. From which we can see that SGcheck detects the out of array boundary error and precisely points out the source of the error.

```
==27103== Thread 45:
==27103== Invalid write of size 4
==27103==    at 0x80641C7: world (test2.c:61)
==27103==    by 0x810758C: process_setup (targint.c:1508)
==27103==    by 0x4DC335A: start_thread (in /lib/libpthread-2.4.so)
==27103==    by 0x4D2BC0D: clone (in /lib/libc-2.4.so)
==27103==  Address 0x2a30f424 expected as actual:
==27103==  Expected: stack array "a" of size 40 in this frame
==27103==  Actual:   unknown
==27103==  Actual:   is 0 after Expected
==27103==
==27103== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 42 from 0)
```

A3.2.5 Output of Rational Purify

The following is the error message extracted from the output of Rational Purify. From which we can see that Purify fails to detect the array overrun error at line 61 and the memory leak

error at line 85, but the leak error at line 85 is debatable, it seems it is an implementation decision to not treat it as a leak error. Anyway, purify detects all the other memory errors and gives precise extra information. But still Rational Purify misreports 30 errors that are not originated from OSE application, due to Purify is capable of suppressing the error at runtime through interaction with users, manually suppressing 30 extra errors won't hurt that much in this case.

UMR: Uninitialized memory read:
** This is occurring while in thread 25390:*
test2_main [test2.c:115]
ose_shell_child [addrem.c:118]
process_setup [targint.c:1508]
start_thread [libpthread.so.0]
clone [libc.so.6]
** Reading 4 bytes from 0xd94ab3dc on the stack of thread 25390.*
** Address 0xd94ab3dc is local variable "sig" in function test2_main.*

ABW: Array bounds write:
** This is occurring while in thread 25390:*
test2_main [test2.c:116]
ose_shell_child [addrem.c:118]
process_setup [targint.c:1508]
start_thread [libpthread.so.0]
clone [libc.so.6]
** Writing 4 bytes to 0x90ba6a8 in the heap.*
** Address 0x90ba6a8 is 41 bytes past end of a malloc'd block at 0x90ba648 of 56 bytes.*
** This block was allocated from thread 25390:*
malloc [rtlib.0]
puri_alloc [heapapi.c:108]
heap_alloc [heapapi.c:611]
zzzmalloc [malloc.c:84]
test2_main [test2.c:113]
ose_shell_child [addrem.c:118]
process_setup [targint.c:1508]
start_thread [libpthread.so.0]
clone [libc.so.6]

ABW: Array bounds write:
** This is occurring while in thread 25390:*
test2_main [test2.c:116]
ose_shell_child [addrem.c:118]
process_setup [targint.c:1508]
start_thread [libpthread.so.0]
clone [libc.so.6]
** Writing 4 bytes to 0x90ba6a8 in the heap.*

** Address 0x90ba6a8 is 41 bytes past end of a malloc'd block at 0x90ba648 of 56 bytes.*
** This block was allocated from thread 25390:*
malloc [rtlib.o]
puri_alloc [heapapi.c:108]
heap_alloc [heapapi.c:611]
zzzmalloc [malloc.c:84]
test2_main [test2.c:113]
ose_shell_child [addrem.c:118]
process_setup [targint.c:1508]
start_thread [libpthread.so.0]
clone [libc.so.6]

MLK: 56 bytes leaked at 0x90ccd90
** This memory was allocated from:*
malloc [rtlib.o]
puri_alloc [heapapi.c:108]
heap_alloc [heapapi.c:611]
heap_alloc_shared [heapapi.c:769]
test2_main [test2.c:114]
ose_shell_child [addrem.c:118]
process_setup [targint.c:1508]
start_thread [libpthread.so.0]
clone [libc.so.6]

A3.2.6 Conclusion

From the test result we can see that OSE kernel cannot detect any error in this case, while the prototype is capable of detecting all the memory errors. However, compared with A2 that can almost perfectly detect all the errors and report precise information, A1 may misreport errors and cannot provide precise extra information for invalid write errors. Rational Purify is equally good at A2 to point out the errors, the extra information and the source of errors, but it cannot identify errors that SGcheck can identify and tends to report OSE core errors.

A3.3 Speed Benchmark

This section benchmarks the speed by running a test case under Rational Purify, Original Valgrind, Customized Valgrind/Memcheck with OSE-WRAPPER A1 and Customized Valgrind/Memcheck with OSE-WRAPPER A2.

A3.3.1 Benchmark Solution

- **Test case:**
  The test case is a modified OSE SFK *refsys* program, which boots up the whole OSE SFK and then terminates itself immediately. The test case is achieved by modifying

the file: refsys/commom/core_supervisor.c, where the following lines of code are added to the end of function: static void core_startup (PROCESS osemain_pid)

```
core_shutdown();
extern void (exit)(int);
(exit)(0);
```

- **Benchmark Tool**
  The Unix "*time*" command is used here to determine the duration of execution of a particular program. "*time*" by default output three items: real, user, sys, which indicate elapsed real time in second, the total number of CPU-seconds that the process spent in kernel mode and the total number of CPU-seconds that the process spent in user mode respectively

- **Benchmark Solution**
  Run the test case itself, under Rational Purify, under original Valgrind/Memcheck, under customized Valgrind/Memcheck with OSE-WRAPPER A1 and A2 respectively. To reach a stable result, repeat the action 20 times and calculate the average value as the result. The reason of choosing Memcheck as the Valgrind Tool for benchmark is that it is a Rational Purify equivalent in terms of functionalities, therefore they are comparable in speed benchmark.

A3.3.2 Benchmark Result

Table A.3.1 gives the benchmark results, from which we can see that both Purify and Memcheck take much more time to analyze the test program than just running the test case. But it is clearly that Purify is the fastest solution to check the memory errors. Although Memcheck is slower, it is however interesting to find that the customized Valgrind/Memcheck with OSE-WRAPPER doesn't consuming much more time than the original Valgrind/Memcheck, especially, the customized Valgrind/Memcheck with OSE-WRAPPER A2 solution takes almost same time to analyze the test case as using original Valgrind/Memcheck.

Table A3.1 Speed benchmark result

Unix "*time*" :	real	usr	sys
Pure test case	3.566	0.006	0.002
Test case under Rational Purify	7.550	2.388	0.202
Test case under original Valgrind/Memcheck	12.680	9.017	0.560
Test case under customized Valgrind/Memcheck with OSE-WRAPPER A1	13.251	9.650	0.576
Test case under customized Valgrind/Memcheck with OSE-WRAPPER A2	12.785	9.221	0.581