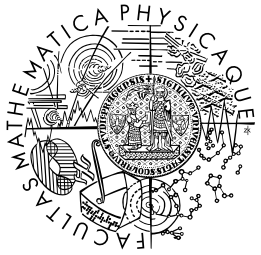

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Roman Kápl

Dynamic Analysis for Finding Endianity Bugs

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Pavel Parízek, Ph.D.

Study programme: Computer Science

Study branch: Software Systems

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Dynamic Analysis for Finding Endianity Bugs

Author: Roman Kápl

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: When two computer systems communicate, for example over a network, they must agree on the ordering of bytes within numbers. This ordering is called endianess. Often one of the systems has to swap the order of bytes to the agreed standard.

Results of this work help programmers to find places in their program where they forgot to swap the bytes. We have developed a dynamic data-flow analysis built upon the popular Valgrind tool. Compared to the static analysis currently used by the Linux kernel developers, our approach does not require annotation of variables with their endianess. Typically only few places in the program source code will need to be annotated. The analysis can also detect potential bugs that would only manifest if the program was run on computer with opposite endianess.

Our approach has been validated on an existing program known to contain yet unfixed endianess problems (RadeonSI OpenGL driver). It has identified all endianess-related bugs and provided useful diagnostic messages together with their location.

Keywords: dynamic analysis, endianess, Valgrind, data-flow analysis

I thank my friend Jan Neuzil and my father for proofreading this thesis and providing a different perspective. I am also grateful to my supervisor, who was genuinely interested both in the technical details of the implementation and the text. And finally, i would like to thank my parents for asking the same question over and over: “When are you going to finish it?”.

I would also like to thank the SYSGO company for providing the PowerPC hardware for testing.

Contents

1. Introduction

Modern computers store integer numbers in memory in two mutually incompatible ways: least-significant byte first (called little-endian) or most-significant byte first (called big-endian). This storage order is referred to as *endianess*, *endianity* or simply *byte-order*. If two computer systems communicate, for example over a network, they must agree which endianess to use, in order to avoid misinterpreting the numeric values of the data.

Processor architectures typically define a *native* endianess, in which the processor stores all its data. Protocol and data storage formats must likewise define an endianess, if they transmit or store integer data in binary. Also devices connected to computers may have control interface with endianess different from the host's native endianess.

Most processors available today have standardized on little-endian, but there are still some processors used in the embedded world (such as 32-bit PowerPC) which are big-endian. The ubiquity of the little-endian x86 architecture also means that PCI(-e) devices are typically little-endian. On the other hand, most network protocols are big-endian (TCP/IP itself is big-endian). Data storage formats are mixed bag, with endianess often depending on their originating platform, and some are even bi-endian, meaning that they have both endianness variants.

This means that programs communicating with other computers, storing data or talking to devices need to swap the bytes inside all numbers to the agreed upon endianess. If programmer forgets to do so, this mistake can often go unnoticed, because programs are developed and tested on little-endian x86. For example, if two identical programs running on little-endian architecture communicate over the network using a big-endian protocol, a missing byte-swap in the same place will go unnoticed. Another example is a device driver communicating with a little-endian PCI device, that does not need any byte-swapping when running on little-endian architecture.

Linux kernel developers are aware of this problem, and therefore they use a static analysis tool that identifies places where byte-swaps are missing (among other problems). The analysis relies on the usage of specialized data-types for all variables with non-native endianess. We describe the approach taken by Linux kernel developers in Section 3.1.

Our work is motivated by a concrete manifestation of this problem. The OpenGL driver for Radeon SI graphics cards for Linux does not work on big-endian machines. There is an interest from a vendor using PowerPC (which is big-endian) to use this card (which is little-endian). Instead of hunting down bugs in foreign code-base, or trying the approach used by Linux kernel developers, we tried a different approach and used the OpenGL driver as a case study (results are described in Section 6.1).

We explore an approach based on dynamic (runtime) analysis. One of our main goals is to reduce the number of the changes needed in the program source code (in comparison with the Linux kernel approach) and to be programmer-friendly where possible. The intended usage is searching for existing endianess bugs, in a similar manner that programmers use Valgrind to uncover undefined behavior and pinpoint its source.

The main idea behind our dynamic analysis is to track endianess of all values in the running program and to report when any data leaving the program has the wrong endianess. The program needs to be modified only to inform the analysis engine when the byte-order is being swapped and when data is leaving the program.

In C programs, byte-order swapping is typically done by system-provided macros, such as `htons`/`htonl` or those defined in the `endian.h` header. Thus only these macros need to be replaced in order to inform the analysis engine about byte-swapping.

Data also tends to leave program only through few functions. For some programs, the appropriate place to check for correct endianness are the `send`/`write` family of syscalls. For other programs, the appropriate place might be a networking library they are using.

The method for endianess tracking we propose is inspired by *taint analysis* (discussed in Section 3.2), which is used to find out how data values propagate inside a program. In our case we are trying to determine how values that are not byte-swapped (and thus have wrong endianess) propagate – in particular, we want to find out if they leave the program.

However, using existing taint analysis tools is not ideal. In Chapter 4 we discuss the deficiencies of the taint analysis in terms of precision and propose improvements. Implementing the analysis ourselves also allows us to provide usability improvements, such as better error reporting and *origin tracking* for values with wrong endianess.

1.1 Target Languages

We have decided to restrict ourselves to discussion of programs in C-like compiled languages (*C*, *C++*, *Ada* etc.), as opposed to higher-level languages like *C#* or *Java*.

While it is true that the analysis might be easier to implement for the higher-level languages, we feel that low-level languages have more pressing need for analysis of this kind. Higher-level languages commonly prohibit treating memory structures as an array of bytes. If program in such a language wants to transfer data (for example over a network), it must do so using a provided abstraction layer, like Java's `DataOutput` interface. These abstraction layers precisely define how the data is stored, including its endianess [Oracle, 2017].

This decision is also consistent with our motivation of improving the state of the Radeon SI driver, which is a implemented as a C library.

We shortly discuss the implications that support for higher-level languages would have for our analysis in Section 8.2.

1.2 Terminology

In the rest of the thesis, we will use the following terms to denote endianess of data in various parts of the system:

- **Native endianess** is the endianess of the architecture on which the analyzed program will run.

- **Target endianess** is the endianess a program should use when communicating with an external entity. This will typically be an endianess of a network protocol, a file format or a connected device. We use this term if it is important to know that the endianess is potentially different from the *native* one, but the discussed analysis does not need to know if the endianess actually is little-endian or big-endian.

- **Concrete endianess** is the endianess the program should use when communicating with an external entity. It is similar to *target endianess*, but we stress that the analysis needs to know if the endianess is little-endian or big-endian.

2. Background – Dynamic analysis

For readers not familiar with dynamic analysis and the common methods used for it, we give a short introduction in this chapter. Feel free to skip it, if you are familiar with the basic principles of dynamic analysis or some of the tools using it, like Valgrind [Nethercote and Seward, 2007], PIN [Luk et al., 2005] or Clang's MemorySanitizer [cla, n.d.] .

Dynamic analysis (as opposed to static analysis) relies on running the program and analyzing its behavior at runtime (as opposed to reasoning about the program without running it). It has the advantage of being able to observe program behavior that is too complex to be reasoned about statically. The downside is that it analyzes only the control-flow paths in the program that were actually executed for the given inputs.

As a simple example, taking a stopwatch and measuring the execution time of a program can be considered the simplest form of dynamic analysis. A more realistic example is sample based profiling: the program is periodically interrupted to find out (sample) which function is currently being executed. If done frequently enough, we can find out which parts of a program are executed most often and therefore represent good candidates for optimization.

But if we want to collect much more data about the program behavior, more heavyweight approach is needed. Often the analysis at hand (as is the case with our endianess-checking analysis) needs to know about each instruction being executed. There are two major approaches, simulation and instrumentation, described in the following sections.

2.1 Simulation

Simulation relies on running the compiled program on a simulated CPU, instead of running it on a real one. The CPU simulator interprets the machine code, replicates the behavior of the original CPU, but also carries out the tasks required by the analysis for each simulated instruction.

For languages utilizing virtual machines (such as Java), there is no need to interpret the machine code, instead it suffices to interpret the VM byte-code – which is what a virtual machine does (if we ignore JIT). Thus often an existing VM can be modified to perform the analysis.

However, interpreting the code usually causes a significant slow-down compared to running the code directly. For this reason, a different technique is often used.

2.2 Instrumentation

Another approach, called instrumentation, is to directly modify the analyzed program to carry out the tasks required for analysis. The modification of the program can be performed at various levels:

- at **source code** level, called source-to-source translation
- at **VM byte-code** level (such as Java byte-code)
- at **compiler intermediate representation** level (such as LLVM IR, used by the Clang compiler)
- at **machine code** level, called binary translation or binary instrumentation

The first option, source-to-source translation, is not very popular for various reasons not discussed here. That leaves us with working on a machine-code, intermediate code or byte-code.

For higher-level languages which are compiled to VM byte-code, most often the byte-code is instrumented. The reasons for doing so are the following:

- Performance – byte-code has typically less instructions than the target machine code, which means less instructions have to be instrumented. Also the VM is able to optimize the additional instructions added during instrumentation.
- More information – byte-code typically preserves more information about the original program structure than machine-code, which allows operations like changing structure layout or function argument list.
- Ease of instrumentation – machine-code can be broken by instrumentation more easily than byte-code.
- Support – instrumenting byte-code is natural choice and libraries are available. Most VMs provide easily accessible hooks to modify the loaded byte-code.

For low-level languages, such as C, usually the machine-code is instrumented. But recently compiler-based instrumentation has been getting more popular, with Clang compiler plugins like MemorySanitizer [cla, n.d.].

Instrumenting the machine-code must be done very carefully, since the compiled machine-code is very fragile and assumes it is run unmodified. Unless the instrumentation is very minor (like instrumenting only function entry and exit). Usually the code is copied to a new location and heavily changed: analysis instrumentation is added and jump targets are updated to reflect new code locations. Indirect jumps are instrumented to compute the new code locations from old code locations and jump to them. Accesses to the program counter register are instrumented to return the old code locations. Old code is preserved at its original location, to maintain the illusion that the code is not instrumented.

2.2.1 Ahead of Time

If program code is instrumented explicitly before being run, this is called *ahead-of-time* (AOT) instrumentation.

Instrumentation techniques based on compiler's internal representation are always ahead-of-time, since they must be run when the program is being compiled.

2.2.2 Just in Time

If program code is instrumented only immediately before being executed, this is called *just-in-time* instrumentation (JIT). For example, a function may be instrumented only when it is called for the first time.

When instrumenting machine-code, this is a typical mode of operation, since it solves some of the problems ahead-of-time instrumentation of machine-code encounters: separating code from data, dynamically loaded code and self-modifying code.

2.2.3 Frameworks

For all of the instrumentation techniques mentioned in this chapter, frameworks to do the heavy lifting are available. This is very useful for machine-code instrumentation, where writing such a framework is a long-time undertaking, due to the number of low-level details that must be taken care of.

The frameworks handle parsing of instructions, make sure the instrumentation can not be noticed by the instrumented code and take care of implementation details. The programmer using these frameworks only has to provide code doing the instrumentation itself.

The individual frameworks suitable for our use-case will be discussed in Section 5.1, among with the advantages and limitations of the techniques they use.

3. Related Work

In this chapter we will take a look at existing work related to our problem of finding endianess bugs. As far as we know, there is only one specialized tool tackling exactly this problem – the Sparse tool used by Linux kernel developers.

We should however note that an approach similar to ours has been briefly suggested on the llvm-dev mailing list [Collingbourne, 2013], to be implemented using the LLVM/Clang plugin called DataFlowSanitizer [dat, n.d.].

As we have mentioned in the introduction, our approach can be considered an extension of the *taint analysis* technique, so we are going to cover it in Section 3.2.

One existing tool that served as an inspiration is Memcheck, part of the Valgrind instrumentation framework (Section 3.3). Memcheck uses an approach similar to *taint analysis* to catch use of uninitialized memory. Also interesting is the runtime type checker Hobbes (Section 3.4), since we can model integers of different endianess as different types (this is indeed what Sparse does).

Taint analysis, Memcheck and Hobbes need to attach additional metadata to memory locations in order to work. This is typically done by a mechanism called *shadow memory*, that will be covered in Section 3.5.

3.1 Sparse (Linux kernel)

Linux kernel developers use a tool called Sparse (shorthand for semantic parser) to provide additional code-checking to the Linux kernel sources. It is comparable to the Lint tool, but detects different types of bugs.

We are interested in how Sparse can provide support for catching endianess bugs. For that we need to understand the `__attribute__((bitwise))` type attribute provided by Sparse.

The attribute is used in definitions of integer types to instruct Sparse that instead of introducing a type alias a new integer type is being defined. The new type will behave mostly identically to the old one, but values of the new type will be incompatible with the values of the old type. Values of different *bitwise* types are not mutually compatible either. Furthermore, arithmetic operations will not be available for the new bitwise type. The bitwise types are documented in [spa, n.d.] and [Brown, 2016].

For practical example, consider the code in Figure 3.1 and the problems reported by Sparse in Figure 3.2. First we define a new bitwise type named `flags_t` (line 3) and then we try following operations with respective results:

- Initialization of a *bitwise* type with a regular integer will fail, even if explicit cast is used (line 5).
- Initialization of a *bitwise* type will succeed when a special `__force` cast is used (line 6).
- Bitwise operations can be used on the *bitwise* type (line 7).
- Arithmetic operations cannot be used (line 8).

```c
#define __bitwise __attribute__((bitwise))
#define __force __attribute__((force))
typedef unsigned int __bitwise flags_t;
static void test(void) {
    flags_t flags = (flags_t) 42;
    flags_t flags = (__force flags_t) 42;
    flags | flags;
    flags * flags;
    int number = (int) flags;
}
```

Figure 3.1: C program demonstrating the behavior of Sparse

```
sparse.c:5:22: warning: cast to restricted flags_t
sparse.c:8:5: warning: restricted flags_t degrades to integer
sparse.c:8:13: warning: restricted flags_t degrades to integer
sparse.c:9:19: warning: cast from restricted flags_t
```

Figure 3.2: Warnings emitted by Sparse for the example code given in Figure 3.1.

- Initializing a regular integer with a *bitwise* type will fail unless the `__force` cast is used (line 9).

The following *bitwise* types are defined in kernel headers to hold integers of a *concrete* endianess: `__le16`, `__le32`, `__le64` (for little-endian integers) and `__be16`, `__be32`, `__be64` (for big-endian). These types are used in structures describing the layout of device interfaces (registers, ring buffers, etc.) or file formats with *concrete* endianness [Corbet, 2006].

The nature of bitwise types ensures that a programmer cannot accidentally assign a regular native endianess integer to a bitwise type or vice versa. It also prevents the programmer from using it in arithmetic operations, which do not make sense on byte-swapped values. Since bitwise operations are preserved under byte reordering, they can be used on the bitwise types without warning.

Macros like `__le32_to_cpu` are provided to safely convert between these *bitwise concrete endianess* integers and regular ones. The macros combine byte-swapping and the necessary `__force` casts. Programmers are expected to use only these macros, thus avoiding errors.

The correctness of the warnings produced by Sparse depends on the diligence of the programmers. They must correctly annotate all structure definitions with bitwise types[1]. If the byte-swapping does not take place immediately before writing to the structure, other places where data with concrete endianess is kept must also be annotated.

We observe that behavior similar to bitwise types can be obtained in C (without Sparse) by defining a single-element structure, since a structure cannot be

[1]Drivers for devices with only registers and no other shared data often have IO functions to access the registers. These functions take care of byte-swapping. In that case, it is hard to make a mistake and omit the byte-swapping.

cast to an integer. The only functional difference between using bitwise types and structures is the unavailability of bitwise operations on structures, thus introducing a certain inconvenience. In C++, an identical behavior to bitwise types could be replicated with a new class and operator overloading.

In the remainder of this chapter, we will look at available dynamic analysis techniques, and how they could be modified and applied to the problem of endianess checking.

3.2 Taint / Data Flow Analysis

Taint analysis is a runtime technique used to track flow of data inside a program. It is described in a summary article on this technique rather well:

> The purpose of dynamic taint analysis is to track information flow between sources and sinks. Any program value whose computation depends on data derived from a taint source is considered *tainted*. Any other value is considered *untainted*. A *taint policy* determines exactly how taint flows as a program executes, what sorts of operations introduce new taint, and what checks are performed on tainted values.
>
> [Schwartz et al., 2010]

An example application of taint analysis is for reverse engineering. In that case, the analysis is performed on machine code of the program, using simulation or dynamic binary instrumentation. One of the program inputs is marked as a taint source and the analysis reveals what parts of the output (or internal state) were influenced by the input. This helps a reverse engineer understand how the analyzed program works.

We propose to adopt this technique for detection of endianess bugs. Tainting can be used to mark native-endianess data. Unless the taint is explicitly removed (by byte-swapping), it will be propagated. *Taint policy* is defined as follows:

- **Arithmetic operations** and **constant data** are *taint sources*. This is based on the assumption that constants are not already byte-swapped.
- **Bitwise operations** and **data manipulation operations** (moves, loads, stores etc.) preserve taint, if operands are all tainted or untainted.
- User (the programmer using the analysis) specifies which functions are **byte-swapping functions**. These functions will accept tainted data and remove the taint. Typical examples are `htonl` and `htons`.
- User specifies which functions are **interface functions**. These functions will check that their arguments are completely untainted.

This setup ensures that native-endianess data cannot accidentally leave a program, if (1) all *interface functions* were correctly marked as such and (2) all functions marked as *byte-swapping* indeed perform correct byte-swapping.

Observe that the analysis can be done even on machines whose native endianess is the same as the target endianess, since tainting can still take place.

If both endianities are the same, byte-swapping functions become identity functions, but the analysis still can remove the taint when data passes through them. This enables us to find data that would not be byte-swapped if the endianities were different.

Moreover, the analysis can be used to detect the opposite direction of errors – programs using non-native endianess data without byte-swapping it first. To detect these errors, arithmetic operations would check that their data is tainted.

The analysis described so far would however run into some problems in practice. The most notable is that byte-sized (`char`) data do not require byte-swapping. Our analysis should therefore have some provision for identifying this data and not considering it as tainted, otherwise it would produce false positives. Identifying the byte-sized data might be easy in higher-level languages, but in C/C++, data is commonly interpreted as different types at multiple points. For example, consider `memcpy` or union types.

We will look at this and other problems in Chapter 4, which extends the tainting policy described so far. Meanwhile, we will discuss other related techniques and tools for dynamic checking of program properties.

3.3 Valgrind

Valgrind [Nethercote and Seward, 2007] is an dynamic binary instrumentation framework for debugging, profiling and analysis. It comes bundled with a set of tools, but also has been used to implement many research analysis tools, for example a taint analysis tool [Newsome and Song, 2005]. We are mostly concerned with the most popular tool that comes with it: Memcheck.

Memcheck [Seward and Nethercote, 2005] is a tool for finding memory leaks[2] and certain types of undefined behavior errors: use-after-free and use of uninitialized variables. For this, Memcheck keeps a valid/invalid flag for each bit of program memory. All newly allocated memory (via `malloc` or stack allocation) is marked as invalid, until the memory is initialized by the program. Valgrind then checks that inputs to operations (like arithmetic operations and control flow operations) are all valid.

The first reason we mention Valgrind here is that what Memcheck does is in practice a form of taint analysis, although Valgrind documentation does not describe it as such. Bits that are marked as invalid are tainted and the behavior described in the above paragraph is the taint policy.

The second reason is that Valgrind is notable for its programmer-friendly nature. In most cases it can be run on complex programs, without recompilation, and detect bugs right away. The problem reports are well presented and include stack-traces.

This shows that a very low-level tool can be made user-friendly. However, replicating fully the "it just works" nature of Memcheck for our endianess checker might prove difficult, because it would require automatic detection of the byte-swapping functions and interface functions (as defined in Section 3.2).

Other tools similar to Valgrind are PurifyPlus, Intel Inspector (based on the Pin framework) and Dr. Memory (based on the DynamoRIO framework). They

[2] The operation of the leak checker is not important for us and is not described in this thesis.

```
==12755== Conditional jump or move depends on uninitialised value(s)
==12755==    at 0x108668: main (vgexample.c:4)
==12755==  Uninitialised value was created by a heap allocation
==12755==    at 0x4C2CEDF: malloc (vg_replace_malloc.c:299)
==12755==    by 0x10865B: main (vgexample.c:3)
```

Figure 3.3: Example Valgrind output with *origin tracking* enabled.

```
1  struct msg {uint16_t id; ... };
2  void prepare_msg(struct msg *m) {
3      m->id = REQUEST_ID_1;
4      ...
5  }
6  int main () {
7      struct msg m;
8      prepare_msg(&m);
9      write(STDOUT_FILENO, &m, sizeof(m));
10 }
```

Figure 3.4: Example program demonstrating the usefulness of *origin tracking* for an endianess checker.

are not as commonly used as Memcheck and they work on the same principles, so we do not need to look further into them.

3.3.1 Origin tracking

Memcheck provides a feature that helps to identify the source of undefined value that was used – and caused Memcheck to report an error [Bond et al., 2007][3]. User (the programmer) gets two stack-traces: one will identify the place when the undefined (invalid) value was used and the second one, provided by *origin tracking*, identifies the source of the value. Example Memcheck report with origin tracking is shown in Figure 3.3. The program examined by Valgrind in the example tried to use freshly `malloc`'ed memory as an `if` condition.

This feature might also be useful for the endianess checker. One stack-trace would show the call to the *interface function*, where the wrong endianess was detected, and the other would show where the value has originated.

For better illustration, consider the program in Figure 3.4. The program sends simple messages to its standard output. Let us assume the message format requires all fields to be big-endian and thus the program contains a bug, because the `m::id` field is not byte-swapped. If we specify `write` as the *interface function*, our analysis will identify the call site at line 9, where the native-endianess data has left the program.

However, the appropriate place for a fix is on line 3, in the `prepare_msg` func-

[3] Author describes the effort as "Ultimately not that useful in practice", but the mechanism has changed and results improved since the publication

Op. 1	Op. 2			
	int8_t	int32_t	pointer	unk32
int8_t	error	error	error	error
int32_t	error	int32_t	pointer	unk32
pointer	error	pointer	error	unk32
int8_t	error	int32_t	pointer	unk32

Table 3.1: Hobbes lookup table for 32-bit addition operation.

tion. *Origin tracking* would provide the user with a second stack-trace, pointing to the the `prepare_msg` function.

For the reasons given above, we have implemented *origin tracking* in our tool. The details are discussed in Section 5.4.

3.4 Hobbes

Hobbes [Burrows et al., 2003] is a runtime type checker for binary programs. It can detect some kinds of type mismatch bugs common in C programs. For example, it guards against using an integer as a pointer or vice versa. It relies mostly on runtime analysis of the binary, but partially also on the debugging information, if present.

In Section 3.1 we have explained that endianess checking in Sparse works by considering integers with a *concrete* endianess as different and incompatible types from regular integers. Let us look at the technique Hobbes uses, and see if we could modify it to track the endianess of integers instead of tracking pointers vs. integers.

Hobbes works by attaching additional metadata to each byte of program memory. The metadata contains the following information:

- The **base type** (Hobbes terminology) of the value stored in that byte. If the type is multi-byte, all bytes are marked with the same type. Examples of types include: pointer, `int8_t`, `uint8_t`, `uint32_t`, `float`, unknown byte, unknown 32-bit word, code and unallocated. Individual pointer types are not distinguished.

- A **continuation** marker. The first byte of each value has this marker unset, and the remaining bytes are set to indicate that they are a continuation of existing value.

- **Initialized** flag to track uninitialized memory. It provides similar functionality to Memcheck.

- **Invariant** flag to mark that the *base type* cannot change. The flag is set if the base type is determined from debugging information. In other cases, Hobbes must assume that a memory location can hold values of different types over time.

During program execution Hobbes checks that (1) all operations consume operands of the correct *base types* and (2) correct *continuation* markers are set

for each byte of the operands as described above. The type information of the result is determined using a lookup table for each operation, which also determines if the operation is valid.

For illustration, a subset of the lookup table is shown in Table 3.1 (taken from [Burrows et al., 2003]). Pointers mentioned in the table are 32-bit. We can see that it prohibits using any arguments of the wrong size and further prohibits adding pointers to pointers (which does not make sense). Adding integers is legal and produces an integer, as is adding integer to a pointer, which produces a new pointer.

Hobbes shows us that using fairly complex metadata is feasible in terms of performance and encoding. In Section 3.2 we have already hinted that an analysis relying on a simple taint bit might have some problems – it will report data that does not need byte-swapping. Tracking word sizes or using continuation bits in similar way Hobbes does might help us solve this problem. Accessing debugging information might also help us to identify byte-sized data (like strings) in constants and further reduce false positive rate. We propose some metadata formats inspired by Hobbes in the Chapter 4.

3.5 Shadow Memory

The dynamic analysis techniques mentioned so far (taint analysis, Hobbes, Memcheck) rely on keeping additional metadata (for example taint bits) for each program variable. The question of how to store and organize this metadata thus arises.

The question is related to the level on which we are performing our analysis. For example, if we would be doing analysis of a Java application, on the level of Java byte-code, we know the structure of the local variables and classes and we can change it to allocate space for the additional metadata. For example, Java Pathfinder [JPF, n.d.] is a JVM that allows the association of metadata (called attributes in their terminology) with Java fields and local variables in this manner.

However, this is not feasible for C or C++ programs, which typically rely on the precise layout of structures. For example, a typical C program would be surprised if each member of `char[]` array would contain additional space for metadata. Thus it is necessary to preserve the original layout of stack and heap.

Valgrind uses an approach that allows it to store metadata for each byte of the analyzed program's virtual address space without changing its layout. This means it is transparent for any C/C++ or machine code program. This technique is called *shadow memory*, and it keeps a translation table (similar to the page tables used by the OS) that maps memory pages to *shadow* pages where the metadata are stored. Like page tables, the translation table is sparse and only describes the areas that the program has actually used, thus allowing the shadow memory to be smaller than the virtual address space it covers. Full description of the technique is given in [Nicholas and Seward, 2007].

4. Design of the Analysis

In Section 3.2 we have proposed to use a variant of *taint analysis* to detect endianess bugs. The proposed analysis relies on tagging each byte of memory as either:

- **Native endianess** – corresponds to tainted data in taint analysis.
- **Target endianess** – produced by functions annotated as byte-swapping.

The difference from commonly used taint policies is that data is tainted by default, to mark native endianess. The reasoning is that data produced by most operations (for example arithmetic) and constants can be assumed to be of native endianess. The analysis checks that no such data leaves the program without being byte-swapped.

However, we have also mentioned that the analysis has certain limitations. As an example, we mentioned false positives on data that needs no byte-swapping, such as byte-sized values.

In this chapter, we will propose ways to extend these simple boolean *native*/*target* metadata into richer formats, which would improve the precision of the analysis. But to do so, we must first look at potential challenges.

4.1 Challenges

This section list the individual problems we must keep in mind when extending our analysis.

Endianess-agnostic data We have already mentioned this problem in Section 3.2. Some data do not need byte-swapping. This typically includes byte-sized numbers (which includes strings). On some more exotic architectures, it might include floating-point numbers, which are stored in non-native endianess on that architecture. The currently proposed analysis would report this data, unless they are manually marked as already byte-swapped by the user (but that would be tedious).

Constant data The problem with constants is two-fold. The first, more serious, is that constants might be endianess-agnostic, but the analysis cannot recognize that, because they are stored in `.data`/`.rodata` program segments as a opaque "soup of data". Hobbes solves this problem by looking at the debugging information to parse the "soup".

The second problem is that constants might contain already byte-swapped data. In that case, the only thing we can do is allow the user to manually mark the data as such. This could be done by providing a compiler attribute to annotate the constants. The annotation would then be embedded in the executable, in a similar way debugging information is embedded there.

We can also ignore the problem and force the user to alway byte-swap the data at runtime.

Literal optimization Sometimes the compiler will initialize an array using an operation wider than the array's element. This makes it difficult to detect element width. Strings are common culprits, for example consider the following code:

```
char x[8] = "DEADBEE";
```

If `x` is a local variable inside a function, the compiler[1] might decide to use a single 64-bit store operation to initialize `x`, instead of initializing it byte-by-byte. This problem only occurs on machine code level, compiler intermediate representation is not yet optimized in this way.

Uninitialized data Some fields inside C structures have specific alignment requirements, which in turn may lead to some space in the structure layout being unused. These empty spaces have undefined content, unless the space where the structure is stored was initialized using `memset` or `bzero` beforehand.

Although this unallocated space is not used and it's endianess does not matter, our analysis cannot know that. Leftover data in this uninitialized space can produce false positives. Another possibility is that the communication protocol deliberately contains unused fields with undefined values.

However, it can be argued that using these structures for communication without zeroing them out first is a bad practice, and it is instead better to report the problems.

Promoted operations Some processor architectures do not offer operations for every width of operands (8, 16, 32 or 64 -bit), but only for the widest variants. This is typical for RISC architectures. Even if the operations are available, such as on x86 architecture, compiler may not use them. Instead, the compiler will promote the operation to the wider variant. This implies that we cannot deduce correct operations sizes from compiled programs – and this might further hinder our approaches to recognize byte-sized data.

The problem would be avoided if the analysis was done on a level of intermediate code, such as the LLVM IR, where the operations are not yet promoted and the operand sizes are correct. But it would be preferable if we would not be forced to commit to LLVM just for this reason[2].

This problem also arises on higher levels than machine code. It is common in C to handle `char`-typed variables as `int` and later down-cast them to `char` when they are finally used in strings or in some other `char`-typed places.

Survive memcpy The metadata should be passed intact through routines like `memcpy`. The challenge is that `memcpy` accesses the data using arbitrary sized units – trivial implementations copy byte-by-byte, optimized ones may use

[1] For example `gcc -O2` v7.3.0 on x64 behaves this way.

[2] Actually, we have discovered this problem only after doing our initial proof-of-concepts with Valgrind (that is without LLVM).

SIMD registers. The solution should not be a replacement of `memcpy` with a version that copies the metadata (as done by Hobbes), but instead should be generic, to be able to cope with user-defined `memcpy`-like routines and other kinds of C trickery.

The analysis proposed so far copes with the problem well, but we should not break this property when extending it.

Byte-shuffling operations (`shift`, bitwise `or`) Apart from simple copying, there are other operations that should also preserve the endianess tags. We will call these *byte-shuffling operations*, since these operations only move around bytes inside larger words.

A simple example of such an operation is bit-shift by multiple of eight bits, in either direction. Another example of these operations is bitwise `or` of two disjoint[3] variables.

The byte-shuffling operations might be both correct and incorrect (from the endianess bugs viewpoint), depending on the context. For example, the following (a bit contrived) code is correct, because the shuffling operation is only used to move the data around and does not change the result:

```
uint32_t data32 = htons(0xDEAD);
data <<= 16; data >>=16;
uint16_t data = data32;
write(stdout, &data, sizeof(data));
```

On the other hand, the following code is incorrect, because the shift operation moves the bytes in memory in different directions depending on the native endianess of the host:

```
uint32_t data32 = htons(0xDEAD);
data <<= 16;
write(stdout, &data32, sizeof(data32));
```

It is not possible to simply tell from code if the usage is correct or incorrect. However, it might be better to choose the more conservative (less false positives) option of letting these operations preserve endianess tags.

Fast handling The metadata should be encoded in such a way that they are easy to handle. Ideally, each instruction in the original program should only have one additional instruction in the instrumented program to track the metadata.

The analysis proposed so far has a single bit and operations either copy the metadata of their operands (when moving the data) or mark the data as *native endianess*. Both metadata operations can be done using single instruction.

[3] By disjoint we mean that `a & b == 0`.

Recognizing byte-swaps So far, we have assumed that byte-swap functions are explicitly annotated by the user, so that we can correctly change the endianess tags. The user experience would be improved, if we could get rid of this annotation and automatically figure out when byte-swap is occurring.

One approach is to do a static analysis of the code and try to recognize sequences looking like a byte-swap. The x86 architecture makes this easier for us, since there is dedicated `bswap` instruction, which is used in optimized code [4]. However, on other architectures compiler optimizations may obscure the byte-swapping and make the sequences hard to recognize.

Another approach is to extend the tagging mechanism to somehow allow us to recognize the result of byte-swap operation. This could be done by storing a byte position for each byte.

However, it is important to note that if the user would choose to use automatic recognition of byte-swaps, it must be run the analysis on machine with the native endianess different from the target endianess, so that the byte-swaps will be present. Because of this limitation, we assume that most users would like to use the annotations anyway.

Usage of target-endianess We mostly focus on ensuring that the program being analyzed does not transmit data of incorrect endianess. There is also the opposite problem: ensure that the program does not use data of other than native endianess.

The analysis described so far allows us to check if all operands of arithmetic instructions have the correct native endianess. While other operations, like comparisons and bitwise operations might be used on any endianess in valid ways, arithmetic operations are unlikely to produce the correct results if fed data of wrong endianess.

One might also experiment with other more strict heuristics to catch more problems: for example comparisons between data of different endianess might be disallowed.

4.2 Proposal 1: Byte-sized Tag

One way to solve the problem of byte-sized data is to simply mark it as such. Thus the possible tags for a byte are: *native*/*target*/*byte-sized*, where *byte-sized* effectively means "endianess does not matter".

But as we have mentioned, knowing the correct size of arithmetic operations is difficult, since operations are often done on wider types than is the actual variable type. This in turn means we do not know if the result is byte-sized or not. There are at least two solutions to this problem, described in the next sections.

4.2.1 Always Mark the First Byte

One simple solution is to always mark the first byte of arithmetic operations as *byte-sized*. This mechanism ensures that if only the first byte of the value is used,

[4] At least for `gcc -O2` *7.3.0*.

it does not trigger any endianess errors when checked, since the memory has a *byte-sized* tag. If the whole value is used (or at least more than just the first byte), there is one *byte-sized* byte and the rest of the bytes are marked as *native*, thus causing an endianess error when checked.

The only drawback to this approach is the slightly inaccurate identification of the memory with wrong endianess in error reports. For example, if we check a `uint32_t` value that has not been byte-swapped, we would expect an error report along the lines of "You have checked memory at location X, bytes **0-3** were not byte-swapped", but instead we get "You have checked memory at location X, bytes **1-3** were not byte-swapped" (first byte is not reported). Furthermore, since the position of the first byte depends on the architecture's endianess, the reports are different on architectures with different endianess.

However, it is possible that the reports could be improved at the time of memory check using a suitable heuristic – for example by taking into account alignment, surrounding bytes, etc.

4.2.2 Guess Operation Width

Another option is to guess the operation widths based on the values of the operands and the results. If all the results and operands are able to fit into a single byte (when they are in the 0 to 255 numerical range), the operation is considered byte-sized.

Notice that this approach will always report superset of the endianess errors when compared with the approach of always marking the first byte. If the heuristic does not fail, the error reports will also identify the correct byte-ranges where non-swapped values are stored.

However, this approach also has potential to misreport endianess errors. For example, consider the following code:

```
uint8_t x = (uint8_t) rand();
check_endianess(x);
```

Since the `rand` function can (and often will) produce values outside the valid range for `uint8_t`, the first byte will not be marked as *byte-sized* and it will be reported as unswapped. The analysis cannot mark it as *byte-sized* at time of the cast, since that would break `memcpy`-like functions, which must preserve metadata.

The other disadvantage is that this approach is computationally more expensive. For each arithmetic operation, we must check the values of inputs and output and choose the correct tag based on that.

Finally, the operands may be signed integers stored as two complement's number. Since operations such as addition are the same for the signed and unsigned variants, we might have to extend the heuristic to also always allow sign extensions of numbers in the -128 to -1 range.

4.3 Proposal 2: Empty Byte Flag

We have mentioned that byte-shuffling operations, like `shift` or bitwise `or` should not change the metadata (tags). For example, the expression `(a & 0xFFFF) | (b << 16)` should result in the metadata of `a` and `b` being concatenated together, since `a & 0xFFFF` and `b << 16` are byte-wise disjoint. We define byte-wise disjointness as: each byte is zero (empty) in at least one of the operands.

One possible solution is to determine whether the operands are disjoint (and computer the resulting metadata) completely at runtime. But in order to improve performance, we can introduce the flag *empty*, signifying that the corresponding byte of memory is zero. These flags will be prepared by other operations in the following way:

- `shift` by amount divisible by 8: mark the newly shifted-in bytes as empty
- Widening (upcast): mark the new bytes as empty
- Bitwise `and`: bitwise `or` of empty flags
- Bitwise `or`: bitwise `and` of empty flags, if disjoint
- Constants: determine from constant value
- Other operations (such as arithmetic): always mark all bytes as non-empty, even if their runtime value is zero.

All these operations on empty flags are simpler than operating with the actual values and thus the scheme has a potential to be faster.

There is also another difference that can be perceived both as an advantage and disadvantage: the empty flag implies that the byte is zero, but the reverse implication does not hold. This is caused by the simplified handling of the arithmetic (and other) operations, which never mark bytes as empty. This handling is not only faster, but disregarding the actual value and instead focusing on the operations makes the empty flag reflect the possible values instead of the actual ones, which might be considered as an advantage[5].

As an example, consider the following code:

```
uint32_t x = (rand() & 0x1FFFF) | (b << 16);
```

The operands of the `|` operator have a one in two chance of being disjoint. Using the empty flag to assess disjointness of the two operands will always correctly guess that they are not disjoint, which better corresponds to the programmers intention.

4.4 Proposal 3: Unknown tag

In order to improve the handling for data where we cannot say if it is already byte-swapped or if it contains only byte-sized data, we can add a third possible tag (or fourth, if we will also use *byte-sized* tag). Let's call it the *unknown* tag.

[5] In retrospect, maybe we have given too much thought to this marginal problem.

We can use this as an "escape hatch" for some of the problems encountered so far: constant data and uninitialized data. Data marked with unknown tag will not be reported as an error if it appears in memory checked for correct endianess (but the behavior can be overridden by the user).

4.5 Proposal 4: Byte Positions

Instead of using the simple native/target metadata, the endianess can be stored explicitly, by tagging each byte in a word with its position. For example of this metadata, see row (1) in Table 4.1. We number the bytes starting from 0 for the least significant byte.

However, this encoding does not suffice to identify the endianness, unless we also know the layout of the data we are checking. For example, consider the sequence of tags $0; 1; 0$. This could be read both as a "byte followed by big-endian integer" and "little-endian integer followed by byte". To avoid the ambiguity, the data must be delimited in some way.

Hobbes type-checker (see Section 3.4 for its description) uses a *continuation marker* for this purpose. All bytes in an integer are marked using a *continuation marker* (a flag in metadata), except the last byte. This way, boundaries between integers can be identified. The use of the *continuation marker* is illustrated on row (2) of Table 4.1. We use $'C$ to stand for the *continuation marker*.

But because of the promotion of operation sizes we have talked about, it is possible that the last byte will be chopped of. The way Hobbes type-checker authors cope with this seems to be: using debugging information and restricting itself to x86/x64 architectures where operation sizes tend to correspond to actual operand sizes. Hobbes also considers `memcpy` and similar functions as "incorrect" from its viewpoint. Type violations that occur inside these functions are ignored.

Since our goal is not full type-checking, we can introduce a more lenient method. We can include the original size of the integer in the metadata, as illustrated on row (3) of Table 4.1.

Our former example with "byte followed by big-endian integer" will be tagged with $0/1; 1/2; 0/2$, while "little-endian integer followed by byte" will now be tagged by $0/2; 1/2; 0/1$. Both cases now can clearly be distinguished.

This scheme allows us do the same checks as with the simple native/target scheme: we can check if data was correctly byte-swapped and if arithmetic operations are not used on byte-swapped data. It also has the advantage of being able to recognize byte-swapped data automatically, with no need for the programmer to provide explicit annotations.

Both parts of the metadata (byte order and word size) can be encoded into one byte. If we ignore SIMD registers, the biggest currently used word size is 64-bits (eight bytes). Three bits are needed to encode the byte position up to the eight byte. There are four possible word sizes (8, 16, 32 and 64-bit) and we need two bits to encode these, requiring six bits in total. Encoding the metadata into a single byte is good for memory consumption but also for performance. If the data and metadata memory locations are the same, metadata operations become easier to implement.

Endianess	Little-endian				Big-endian			
Byte	0	1	2	3	0	1	2	3
Data	`EF`	`BE`	`AD`	`DE`	`DE`	`AD`	`BE`	`EF`
(1) Byte-order	0	1	2	3	3	2	1	0
(2) With continuation	$0'C$	$1'C$	$2'C$	3	3	$2'C$	$1'C$	$0'C$
(3) With size	0/4	1/4	2/4	3/4	3/4	2/4	1/4	0/4

Table 4.1: Possible metadata schemes for a 32-bit integer `0xDEADBEEF`.

4.6 Implemented Meta-data Model

We have implemented the model based on the simple tagging option, with addition of proposals 1, 2 and 3. We have decided against the more complex tracking of byte positions (proposal 3), if the tool works satisfactorily without it. Compared to tracking the byte positions, the only known disadvantage of the chosen solution is its inability to automatically recognize byte-swaps.

Each byte of memory and processor registers have one of these tags:

- **Native**: The default endianess produced by arithmetic and other operations.
- **Target**: Endianess used for data explicitly marked by the programmer.
- **Byte-sized**: Endianess to mark the first byte of a word.
- **Unknown**: Endianess for unknown data – constants, newly allocated heap areas, newly allocated stack.

In addition to these tags, each byte can be marked by the **empty** flag, signifying that the byte's value is zero. Operations propagate the metadata tags in the following way:

- **Move** (incl. mem read/write): copy the metadata without change.
- `shift` (by multiple of 8): shift the metadata, newly introduced bytes are marked as *native* and *empty*.
- `shift` (other): handle as an arithmetic operation.
- **Narrowing cast**: copy the part of metadata without change.
- **Widening cast**: copy metadata, newly introduced bytes are marked as *native* and *empty*.
- **Bitwise `and`**: if tags are the same for both operands, propagate them, otherwise use tag rules for arithmetic operations. In both cases, *empty* flags are or-ed together.
- **Bitwise `or`** (disjoint non-*empty* flags): Propagate the tags from the operand where the byte is marked non-*empty*. If both bytes are empty, mark as *native* and *empty*.
- **Bitwise `or`** (otherwise): handle as a standard bitwise operation (see below).

- **Other bitwise operations**: if tags are the same for both operands, propagate them and do not mark any bytes as *empty*. Otherwise handle like a standard arithmetic operation.

- **Constants** (immediates in code): tag first byte as *byte-sized* and the remaining bytes as *native*. Leading zero bytes are marked as *empty*.

- **Arithmetic and any other operation**: mark the first byte as *byte-sized* and the rest of the bytes as *native*. No bytes are marked as *empty*.

The only way to create data with *target* tag is via explicit annotation from the user.

If the user/programmer requests to check a piece of data for correct endianess, the analysis verifies that the data contains no *native*-tagged bytes. *Target* and *byte-sized* tags are allowed and user can choose whether the *unknown* tag should be reported as an error.

4.7 Limitations

It should be clear from the previous sections that our analysis algorithm relies on some guesswork and user cooperation. For example, consider the following pseudocode, which illustrates the inherent limitations of our approach:

```
uint16_t x = htons(0xBEEF);
encrypt_aes(KEY, &x, sizeof(x))
decrypt_aes(KEY, &x, sizeof(x));
check_endianess(x);
```

It is unlikely that the metadata could survive any complex encryption, without the user explicitly telling the analysis that the metadata of `x` are the same after the encrypt/decrypt transformation. The reader can surely think of similar identity transformations – like compress/decompress – that the metadata won't be able to survive.

If the data is leaving the program compressed/encrypted or otherwise encoded, the user should simply request an endianess check before the compression. If the compression/encryption protocol also contains endianess dependent data, the check can be requested twice.

Our approach also does not support programs communicating using protocol with two different *target* endianess at the same time. However, the extension to support this would probably be straightforward, by maintaining two *target endianess* tags. Our current recommendation is to test these two endianities separately (two runs, two compilations).

The analysis is also not able to recognize the endianess of constants, and so they must always be annotated if they are already of the correct endianess. This includes constants stored in data sections of executables and immediate values in instructions.

4.8 Protected Memory

So far, we have assumed that data will be checked for correct endianess on explicit request of the user/programmer. This works if the program uses library functions and/or system calls to move the data from the program.

However, during our tests with the Radeon SI OpenGL driver (described in Section 6.1), we have noticed that the driver directly maps device memory into the user-space process and thus there is no single point where to check the endianess[6]. If the data leaves a program through a memory mapping, without any obvious "check-point", the analysis must instead check all writes to the memory-mapped region.

To solve this problem, we introduce a new kind of annotation. It is used to mark a region of memory as *protected* – meaning that all writes to such region will automatically check the endianess.

To implement the memory protection, each byte of protected memory is tagged with a flag, so that we can find quickly if memory is protected. Please note that this flag is associated with *memory location* while the endianess tags discussed previously are associated with *values*.

In practice, both types of metadata can share the same storage, but operations must be careful not to copy the *protected* flags to other parts of memory or clear it. Manipulation of the protected flag is done only explicitly by the user/programmer when marking memory region as protected.

4.9 Lessons Learned

Note: This section only discusses the high-level analysis algorithm and gives further rationale for its design. For experience with implementation, see Section 6.3.

For our implementation, we have decided to start with the simpler model, extended with the *byte-sized* and *unknown tags* (Proposal 1, Section 4.2, and Proposal 2, Section 4.3 respectively).

Initially, we have not realized the problem with operations being promoted to wider ones. To remedy the problem with promoted operations, we have used a limited version of the heuristic based on the actual value of data, but applied only to constants. This means constants had their size guessed from their values and other operations used the original operation size without any guessing. This has fixed functions like `memset`, which were broken without this heuristic.

Later on, we have realized it might work to always mark the first byte as byte-sized and adopted this approach. The previous approach has worked so far, partly because we used the x64 architecture for our tests and partly because we performed most of the tests on the Radeon SI driver, which uses mostly `uint32_t` and `float`, but no byte-sized integers.

Another extension was the support for shift and bitwise or operations, supported by the *empty byte* flag mechanism (Proposal 3, Section 4.4), implemented in an effort to make the analysis more complete and resilient (but without any real use-case in the Radeon SI code driving this).

[6]Later we found there is a point where ownership of the buffer is explicitly transferred to the device and the check can be performed there. But the feature we will describe is still useful.

To test the problematic parts where our approach might fail, we have built a small set of test cases before fixing the problems. The test cases are located in the source code tree under `endicheck/tests` (see attachment 1) along with the expected error report for each one.

5. Implementation

For implementing the analysis algorithm described in Section 4.6, we have to instrument the program provided by the user with additional instructions to track the endianess metadata. The most important part for that is choosing the right instrumentation framework (Section 5.1). Then we will look at our implementation of *shadow memory* in Section 5.3 and *origin tracking* in Section 5.4.

5.1 Instrumentation framework

The choice of an instrumentation framework is important for several reasons. First, it can help us cut down the development time. It will also affect the way users can interact with our tool. The frameworks have varying support for operating systems and processor architectures. Some of them might also require recompilation of analyzed programs, including all used libraries, which would be inconvenient for the user. Lastly, they also affect the runtime speed of instrumented programs. We will review the most common available frameworks and discuss our choice in this chapter.

We can roughly divide the currently available instrumentation frameworks suitable for C-like languages into two categories – **dynamic binary instrumentation** and **compiler plugins**. *Compiler plugins* are a way to perform the instrumentation during program compilation, as an extra compilation phase. *Dynamic binary instrumentation* tools instead take a program compiled the regular way and instrument the machine code at runtime.

From the user's point of view, using a dynamic binary instrumentation based tool would be easier, because she/he can use the same program for both the analysis and regular runs without recompilation. Also it does not force the user to choose a concrete compiler.

For the analysis to work correctly, all components that handle the byte-swapped data (libraries and the main program) must be instrumented. This includes the C library – the program will likely be using `memcpy` and similar functions. If some components are instrumented and others are not, we should ensure that the program will at least run, even though the analysis results are going to be inaccurate. To guarantee this, we must not change the application binary interface (ABI) of the instrumented components, thus making the analysis more complex. For example, we might be inclined to pass metadata during function calls as additional arguments – but this changes the ABI. The approach taken by Clang's DataFlowSanitizer to this problem is detailed at the end of Section 5.1.1

Since dynamic binary instrumentation based tools instrument all code loaded into the process, the ABI problem does not arise when they are used.

One point worth mentioning is also the ability to instrument kernel code – where a lot of device drivers that would benefit from this tool live. Using dynamic binary instrumentation for kernel code is possible (see, for example PinOS [Bungale and Luk, 2007] or the QEMU variant TEMU [Song et al., 2008]), but it relies on system-emulation and so it cannot be used directly on real hardware. Implementations used in practice, such as Linux kernel's memory checker KASan are currently implemented as a compiler plugin.

5.1.1 Clang/LLVM

Clang is a C/C++ compiler based on the LLVM compiler infrastructure. LLVM supports pluggable compilations passes, which can be used for code instrumentation (among other things). There already exists a mature tool for LLVM, called MemorySanitizer [cla, n.d.]. It provides Memcheck-like functionality and thus shows that the instrumentation has the features we need.

Even better, there exists a plugin called DataFlowSanitizer [dat, n.d.], providing a generic data-flow analysis framework. Even though the framework can not be used as-is (it propagates metadata differently than we want to), it can be used as a starting point. It already provides shadow memory implementation and a way to avoid ABI change for instrumented components. It does not currently provide origin tracking.

Even if the ABI change problem is solved, this only ensures that analyzed programs won't crash, but the correctness of the metadata tracking is still not guaranteed. DataFlowSanitizer solves the problem by requiring the user to provide annotations to functions in uninstrumented components. These annotations describe what effect the functions have on metadata – for example `memcpy` simply copies the metadata. It is unknown how many of these annotations would be needed for average programs to work.

For comparison, MemorySanitizer solves the problem this way:

> MemorySanitizer requires that all program code is instrumented. This also includes any libraries that the program depends on, even libc. Failing to achieve this may result in false reports. For the same reason you may need to replace all inline assembly code that writes to memory with a pure C/C++ code.
>
> Full MemorySanitizer instrumentation is very difficult to achieve. To make it easier, MemorySanitizer runtime library includes 70+ interceptors for the most common libc functions. They make it possible to run MemorySanitizer-instrumented programs linked with uninstrumented libc. For example, the authors were able to bootstrap MemorySanitizer-instrumented Clang compiler by linking it with self-built instrumented libc++ (as a replacement for libstdc++).
>
> [cla, n.d.]

LLVM (and thus Clang) today supports a variety of operating systems (including Windows and Linux) and architectures. However, MemorySanitizer and DataFlowSanitizer currently only support Linux, according to their documentation.

Kernel code instrumentation is also possible with Clang, demonstrated by Linux KASan – Kernel AddressSanitizer. Traditionally, the Linux kernel supported only compilation with GCC, but a progress has been made to support Clang [ker, n.d.]. However, the support is a recent addition and there are still issues.

5.1.2 GCC

The GCC compiler provides AddressSanitizer, an instrumentation based tool designed to catch use-after-free errors. This is a subset of what Clang's Memo-

rySanitizer provides – Clang also checks for the use of uninitialized values.

GCC supports greater variety of operating systems and architectures compared to Clang. Support for Windows is provided by the MinGW port, but its AddressSanitizer support is not yet ready [MinGW team, n.d.].

Linux kernel's KASan was originally developed for GCC and is supported out-of the box.

GCC is not as open as LLVM, which is designed to be used as a library from the start, and therefore is fairly extensible and better documented. Further research would be needed to find out if *GCC*'s plugin API is sufficient to write the instrumentation or if we would need to modify the compiler itself.

5.1.3 Valgrind

Valgrind [Nethercote and Seward, 2007] is a well established dynamic binary instrumentation framework, best known for the Memcheck tool[1].

Valgrind works by first translating machine code that is about to run to an internal machine-independent representation called Vex IR. Vex IR is instrumented and translated back to machine-code. This approach provides architecture independence. Valgrind itself supports several architectures, but its operating system support is limited to Unix-like operating systems (e.g. Linux, Solaris but not Windows).

The performance is lower due to this translation, but the slowdown is tolerable – about 20 times for Memcheck reported by [Nethercote and Seward, 2007]. We expect similar slowdown for our tool.

Apart from providing the instrumentation mechanism itself, Valgrind also provides command-line handling, error reporting, error deduplication and other helper functions. This not only cuts down the development time, but also provides a familiar interface to the user of the tool.

5.1.4 Pin

Pin [Luk et al., 2005] is a dynamic binary instrumentation framework developed by Intel. It is used by Intel for their tools like VTune (performance analysis) and Intel Inspector XE (similar to Memcheck). Since these tools are not open-source, they prove the maturity of Pin and its fitness for our purpose, but cannot be reused as basis of our tool or as inspiration.

Pin does not use any intermediate representation, but instead allows the analysis to instrument the code directly. Pin provides helper functions for the instruction parsing and manipulation. This makes the tools built using Pin architecture-dependent, unless they do simple instrumentation that does not need to know about concrete instructions. However, Pin currently only supports the x86 and x64 architectures, so it does not really matter. Pin supports Windows and Linux, but lacks support for other Unix-like systems.

Another problematic point is the Pin license, which does not allow redistribution without Intel's permission. The license is acceptable for research work, but would limit the usefulness of the tool for end users.

[1] The term Valgrind is often used to refer to the Memcheck tool itself, since it is the default tool run when the `valgrind` command is invoked.

[Nethercote and Seward, 2007] contains a feature comparison between Pin and Valgrind and points out some other features missing in Pin, mainly with regard to support of shadow memory. But it is possible that the issues have been fixed since the date of the comparison.

5.1.5 DynamoRIO

DynamoRIO [dyn, n.d.] is a dynamic binary instrumentation framework, originally developed for runtime optimization. It comes with a Memcheck-like tool, Dr. Memory.

DynamoRIO is similar to Pin in the way instrumentation works: directly on machine-code, without any architecture-independent intermediate representation.

DynamoRIO supports the x86, x64, ARM and AARCH64 architectures. Note that all of these architectures are little-endian (which is a bit unfortunate, given our focus on endianess) and PowerPC is not supported (note that we want to fix the Radeon SI driver on PowerPC).

5.1.6 Our Choice

Each instrumentation framework has its advantages and disadvantages. In our opinion, none of them is a clear winner. We have settled for Valgrind, with its architecture-independence provided by the Vex IR, broad range of supported architectures and the familiar user interface provided to the user. Also we can use the Memcheck source code as a reference, since it is in some ways similar to the tool we are developing. Valgrind is also licensed under GPL, which is an acceptable license for us, because it does not limit the redistribution of the finished tool.

On the other hand, it lacks Windows support and it cannot instrument kernel device drivers. Valgrind's performance is also not the best among the reviewed choices.

5.2 Project Organization

Our tool is distributed as a fork of the Valgrind source code. Technically, Valgrind tools can be distributed as separate projects, but distributing our tool as a fork allows us to make changes to Valgrind itself and use its build and test infrastructure. The source code is attached to this thesis (attachment 1). Binaries for selected Linux distributions are also available (attachment 2).

Our tool is named Endicheck, following the name of Memcheck. If the user has installed our modified version of Valgrind, it can run Endicheck by executing the following command:

```
valgrind --tool=endicheck [OPTIONS...] PROGRAM ARGS...
```

The tool's code is located under the `endicheck` directory of the source code tree. It is organized into the following parts:

- `ec_main`: tool initialization, command-line handling and top-level Vex IR translation routines
- `ec_errors`: error reporting, formatting and deduplication
- `ec_shadow`: shadow memory handling, for storing the endianess metadata, protection status and origin tracking information
- `ec_util`: utility function for general use and for metadata manipulation
- `endicheck.h`: public API with annotations accessible to programs (called client requests by Valgrind)

The entry point into Endicheck is the `EC_(pre_clo_init)` function in the `ec_main.c` file. And yes, `EC_(pre_clo_init)` is a function name, it is Valgrind's convention to use wrapper macros like `EC_` or `VG_` to prefix identifiers and break most IDE's auto-completion.

The heart of endicheck itself and the best place to start diving into the code is the `EC_(instrument)` function (also in `ec_main.c`). Valgrind calls this function each time a new chunk of code needs to be instrumented.

Code is represented by an `IRSB` (intermediate representation super-block) – a block of Vex IR code with a single entry point and multiple possible exit points. The Vex IR itself is composed of temporary variables, processor registers, statements, and expressions (part of the statements). Unfortunately, Vex IR is not really well documented, the best starting point for learning is probably the `VEX/pub/libvex_ir.h` header.

A short example of a Vex IR corresponding to two x86 instructions is shown in Figure 5.1. Valgrind can be instructed to dump instructions and Vex IR in this way by passing these options: `--trace-flags=10000000` (show front-end Vex IR) and `--trace-notbelow=0` (use these settings for all code).

Lines 1 and 7 of the example show the original x86 `mov` instructions. Each instruction is translated into a sequence of Vex IR statements, connected through temporary variables like `t0`. The `GET` expressions and `PUT` statements are used to access processor registers. Register 8 is `eax` and register 68 is the instruction pointer, which is updated explicitly in Vex IR.

The `instrument` function receives `IRSB`s from Valgrind and builds new instrumented `IRSB`s. Each Vex IR statement in the input is copied to the output as-is, to preserve original behavior of the program. But before each original Vex IR statement are the new injected statements responsible for the metadata tracking. The metadata tracking statements are generated by a group of functions with the prefix `shadow_`, each one handling one statement type. If origin tracking is enabled, each temporary variable will also have two "shadow" couner-parts created, one to hold the endianess, and the second to hold the origin tracking information.

Each expression also receives it's shadow counterpart. Functions with the suffix `2shadow` are used to transform original expression into a pair of expressions, one operating on the endianess metadata and one on the origin tracking information. The starting point for the translation of expressions is `expr2shadow`, implementing the algorithm detailed in Section 4.6, with the help of other `2shadow` functions handling concrete expression types.

```
1  0x8048074:  movl $0xDEAD,%eax
2
3      ------ IMark(0x8048074, 5, 0) ------
4      PUT(8) = 0xDEAD:I32
5      PUT(68) = 0x8048079:I32
6
7  0x8048079:  movl %eax, 0x804908A
8
9      ------ IMark(0x8048079, 5, 0) ------
10     t0 = 0x804908A:I32
11     STle(t0) = GET:I32(8)
12     PUT(68) = 0x804807E:I32
```

Figure 5.1: Example x86 assembly translated to Vex IR.

```
1  ------ IMark(0x8048075, 5, 0) ------
2  PUT(360) = 0x5050103:I32
3  PUT(8) = 0xDEAD:I32
4  PUT(420) = 0x1010103:I32
5  PUT(68) = 0x804807A:I32
6  ------ IMark(0x804807A, 5, 0) ------
7  t18 = 0x5050103:I32
8  DIRTY 1:I1 :::
   ↪  ec_store_ebit_32[rp=2]{0x5800b1c0}(0x8049090:I32,t18)
9  STle(0x8049090:I32) = 0xDEAD:I32
10 PUT(420) = 0x1010103:I32
11 PUT(68) = 0x804807F:I32
```

Figure 5.2: Vex IR assembly instrumented by Endicheck.

An example of Vex IR code instrumented by Endicheck can be seen in Figure 5.2. This code is an instrumented version of the code in Figure 5.1. There are slight changes in the statements coming from the original, for example lines 10 and 11 in Figure 5.1 were merged into line 9 in Figure 5.2. These are optimizations done by Valgrind and do not change the semantics of the code – so please ignore them when reading. Also ignore lines 7 and 8 for now, they will be explained in the following section.

The lines added by Endicheck (2, 4, 7, 8 and 10) are highlighted. They are responsible for maintaining the metadata. For example, line 2 stores the metadata `0x05050103` into a shadow register 360. Byte `0x05` represents the empty + native-endianess tag, `0x01` represents the native-endianess tag and `0x03` represents the byte-sized tag. This corresponds with the fact that integer `0xDEAD` has first two bytes empty and that all bytes in constants are marked as native-endianess, except the first byte, which gets byte-sized tag. Complete metadata rules are defined in Section 4.6.

5.3 Shadow Memory

The shadow memory implementation is very closely based on the model that Memcheck uses (slightly outdated description is in [Nicholas and Seward, 2007]). Unfortunately, it is not possible to use the Memcheck code directly, as it is closely tied in to the metadata format used by Memcheck. As an example, Memcheck's shadow memory code uses a technique called *V-bit compression* to reduce memory usage in the common cases, where all bits in a byte are either uninitialized or initialized. This technique is not applicable in Endicheck, since our analysis does not keep per-bit metadata.

Our shadow memory code resides in the `ec_shadow` file and is called from the `ec_main` when memory load/store is encountered. It emits Vex IR expressions and statements to load/store metadata associated with a requested memory location. These Vex IR statements are also responsible for providing the protected memory mechanism as described in Section 4.8.

An example of code generated by `ec_shadow` can be seen on lines 7 and 8 of Figure 5.2. It uses Valgrind's *dirty call* mechanism, which allows us to call C functions from Vex IR. The function called here, `ec_store_ebit_32` is responsible for storing a 32-bit sized metadata to its correct location.

For locating the metadata, a sparse map, similar to a single-level memory-translation table, is used. It maps each 64 KiB page of memory to a different page holding the endianess metadata. If metadata for a yet unseen page is requested, a new page is allocated and all bytes are tagged with *unknown* tag. The mapping is illustrated in Figure 5.3.

Covering the whole 64-bit virtual address space, or even the canonical address space actually usable by the process on x64[2], is impractical. It would require either large translation tables or introducing additional translation table levels. For this reason, the map covers only the first 128 GiB of address space, which need only 2^{21} entries.

The rest of address space (if running on a 64-bit architecture) is covered using a hash-map (called `OSet` in Valgrind). This is method is slower, but Valgrind tries to push all virtual address space allocation below 128 GiB to mitigate the slowness. The mapping for large addresses is illustrated in Figure 5.4.

5.4 Origin Tracking

Our implementation of origin tracking is a bit different from Memcheck, which uses hash-maps to store origin tracking information for memory locations.

Memcheck uses two hash-maps, the first hash-map acting as a cache for the second one. The reasons for this setup are a mix of historical evolution and performance. In Memcheck, only uninitialized values can cause error reports and so only they need origin tracking. The assumption is that the uninitialized values will be a minority in a typical program and Memcheck will benefit from the sparse storage in a hash-map[3].

[2] 48 or 56 bits at time of writing this thesis.

[3] This is our best guess, the actual reason is unclear. The historical reason is that originally only the first level map was present, with limited number of slots. The origin tracking was thus effectively only best-effort.

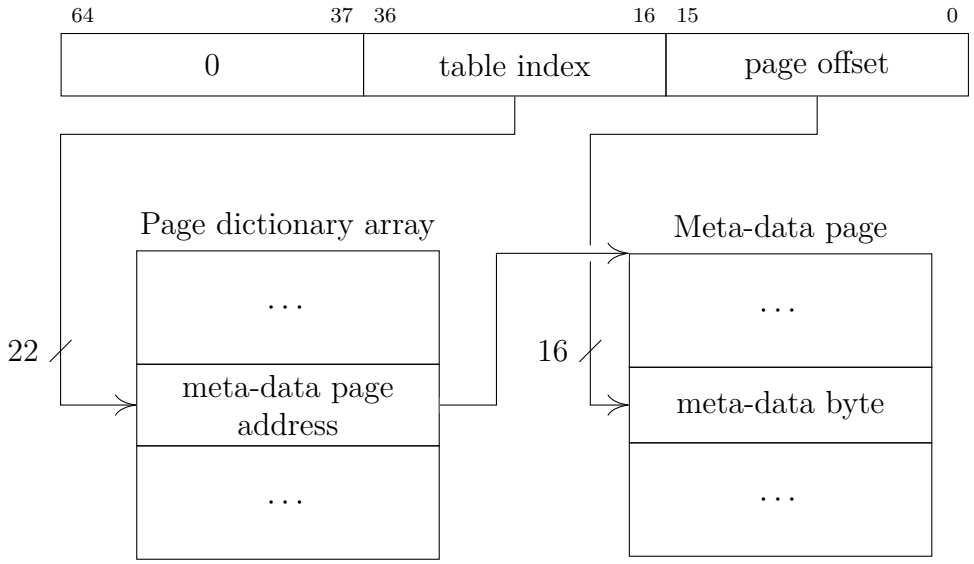

Figure 5.3: Meta-data storage for virtual addresses < 128 GiB

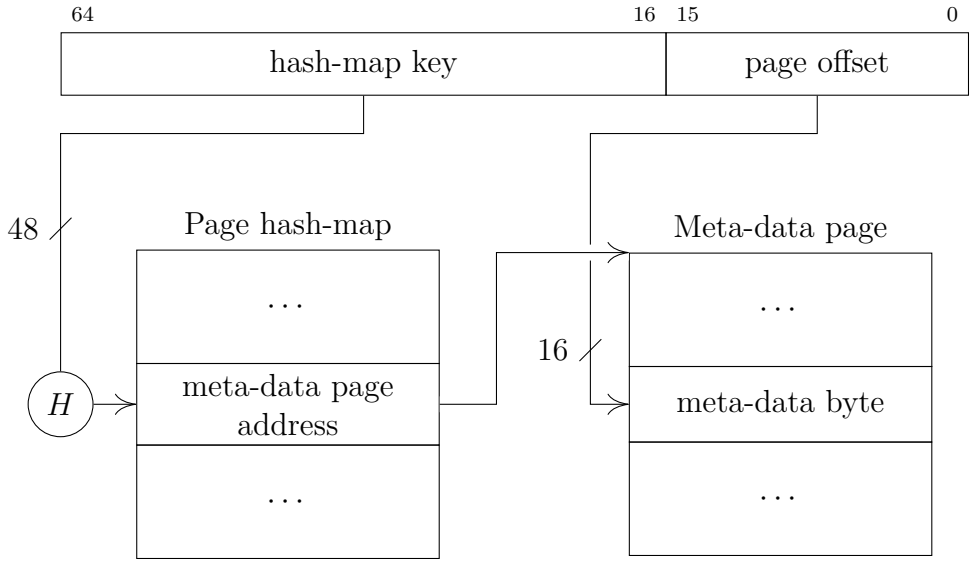

Figure 5.4: Meta-data storage for virtual addresses ≥ 128 GiB

Situation in Endicheck is different, because all values (except byte-sized ones and byte-swapped ones) need origin tracking, since they can be source of errors. Thus the two levels of hash-map is not as good fit and we have instead decided to store the origin information alongside the other metadata in shadow memory. This solution is also simpler, since the same code paths can be used both for metadata and origin tracking.

The origin-tracking information is represented as a 32-bit integer, an execution context identifier provided by Valgrind. The identifier can be converted back to a stack trace. To make the storage more effective, each two bytes of memory share one identifier. Memcheck has similar optimization (each four bytes share an origin information). We have chosen the 2:1 ratio, since all data with wrong endianity must be at least 16 bits wide.

Origin tracking also presents one additional difficulty: the information must be kept for each of processor registers, which can be smaller than the 32-bits required for the execution context identifier. Valgrind stores the processor registers in a so called *guest area* and their offsets inside this area are used as register identifiers by Valgrind. The layout for origin tracking area must be different from the guest area layout, since the register sizes are different.

Originally, the mapping between these layouts was done by code in the file `mc_machine.c` in Memcheck. We have moved this file to Valgrind core so that both Endicheck and Memcheck can use it.

The current disadvantage is the fact that even large types (even SIMD ones) get a single origin tag, mainly due to shadow area design. Since the origin tags are only user convenience, this is not so critical, but should be improved in the future.

6. Evaluation

It is difficult to find objective criteria for evaluating the Endicheck tool we have developed. There is no directly comparable tool and no established corpus of programs with endianess errors available for evaluation.

Instead, we have used the Radeon SI driver as a case study and documented our experiences with fixing the driver and leave the reader to form his own opinion.

One aspect we can measure, however, is the performance of Endicheck and we will give a quick comparison with uninstrumented programs and other Valgrind tools.

We would also like to use this chapter to share a bit of subjective experiences with Valgrind, for the benefit of other people planning to use it for similar purposes.

6.1 Case Study

Radeon SI is a Linux OpenGL driver for a range of Radeon graphics cards, starting with the SI (Southern Islands) line of cards and continuing to the currently produced models. Since these Radeon cards are little-endian[1], the driver must byte-swap all data when running on big-endian architecture such as PowerPC. Because the Radeon SI driver does not currently do that, it simply does not work in that case – it either causes GPU crashes or crashes of OpenGL programs.

6.1.1 Driver Introduction

To better understand the errors we have found, let us first explain how the RadeonSI OpenGL driver fits with the surrounding infrastructure. The first important thing to clear up is that the RadeonSI driver is a user-space driver, not a kernel driver. It is loaded as a shared library into programs that use the OpenGL library (`libGL.so`). The situation is illustrated in Figure 6.1. A kernel driver named `radeon` (without the SI suffix) also exists but has different responsibilities and the two should not be confused.

The Radeon SI driver itself is a part of the Mesa project. The project provides lot of functionality common to all OpenGL drivers. To further simplify development, drivers for modern graphic cards can be built on the Gallium3D infrastructure. Drivers built in this way export a Gallium interface, describing the basic functionality of the graphics card – buffers, commands and shader programs. A so called Gallium OpenGL state tracker, part of the `libgallium` library, is responsible for translating between the complex OpenGL API and simpler Gallium interface. The component is called state tracker, because OpenGL is stateful, while the Gallium interface is stateless.

The whole Mesa machinery is loaded by the `libGL` dispatch library[2]. Mesa then loads the appropriate driver depending on the graphics card installed in the system. In our case, the `radeonsi` driver is loaded, which is linked with

[1] Generations before Southern Islands could be switched to big-endian mode (they did byte-swapping in hardware) and were supported by the r600 driver on big-endian hardware.

[2] On older systems `libGL` is provided directly by Mesa, without any dispatch layer.

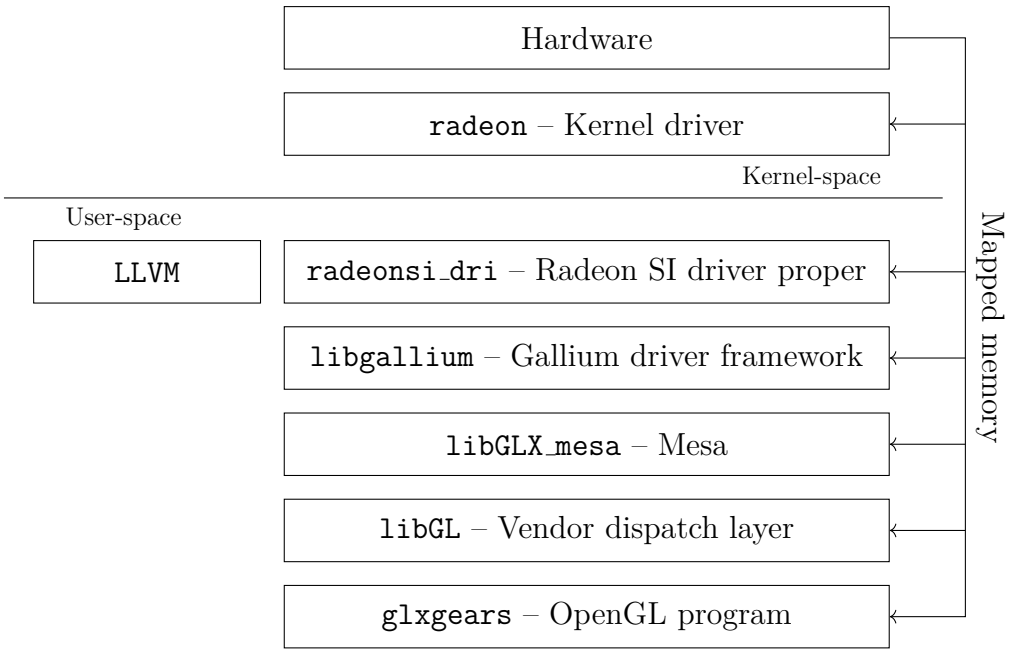

Figure 6.1: RadeonSI driver in the context of a typical Linux OpenGL stack.

the Gallium library. The Radeon SI driver also relies on the LLVM compiler infrastructure to compile shader programs to the instruction set used by the graphics card.

The driver cannot access the graphics card directly, it must go through the kernel driver. However, the kernel driver is less complex and is not the focus of our analysis. It already works on big-endian platforms, as the kernel standards are higher in this regard.

The whole stack communicates not only using function calls and syscalls, but it also makes extensive use of mapped memory. The shared memory is a part of all layers of the stack, including the user program. OpenGL allows mapping the device memory directly into the user process using calls like `glMapBuffer`.

For our analysis, the design of the stack has two implications. First, we have to correctly handle the flow of data through the whole stack and instrument all the libraries used. Second, we must deal with the shared memory – for which we have added the *protected memory* feature.

6.1.2 Driver Instrumentation

One of our stated goals was to reduce the number of annotations that need to be added by the user. The user needs to add two types of annotations: one to byte-swapping functions and one to inform Endicheck that data is leaving a program.

All byte-swapping functions used by Gallium drivers are located in a single file: `src/gallium/auxiliary/util/u_math.h`. This was the only place where byte-swapping functions needed to be annotated. Roughly 20 lines were changed or added.

After some experiments, we also found a suitable function where to check endianity. We have annotated the `radeon_drm_cs_add_buffer` function. To do

```
Thread 9 gallium_drv:0:
Memory does not contain data of Target endianity
Problem was found in block 0x41BF000 (named radeon_add_buffer) at
↪  offset 0, size 8:
   0x41BF000: N N N N N N N N
The value was probably created at this point:
   at 0x8B787F7: si_init_msaa_functions (si_state_msaa.c:94)
   by 0x8B4F979: si_create_context (si_pipe.c:279)
   by 0x8B4FE50: si_pipe_create_context (si_pipe.c:442)
   by 0x88C12F9: st_api_create_context (st_manager.c:883)
   by 0x8A1DACE: dri_create_context (dri_context.c:140)
   by 0x8A1CF8B: driCreateContextAttribs (dri_util.c:479)
   by 0x4C76647: dri3_create_context_attribs (dri3_glx.c:298)
   by 0x4C766E3: dri3_create_context (dri3_glx.c:329)
   by 0x4C463AA: CreateContext (glxcmds.c:304)
   by 0x4C46661: glXCreateContext (glxcmds.c:427)
   by 0x10B67A: make_window.constprop.1 (glxgears.c:559)
   by 0x109A86: main (glxgears.c:777)
The endianity check was requested here:
   at 0x8B85C45: radeon_drm_cs_add_buffer (radeon_drm_cs.c:375)
   by 0x8B4A58B: si_set_constant_buffer (r600_cs.h:74)
   by 0x8B708D0: si_set_framebuffer_state (si_state.c:2934)
   by 0x8AB3CE2: tc_call_set_framebuffer_state
   ↪  (u_threaded_context.c:594)
   by 0x8AB20CF: tc_batch_execute (u_threaded_context.c:96)
   by 0x898037D: util_queue_thread_func (u_queue.c:271)
   by 0x8980016: impl_thrd_routine (threads_posix.h:87)
   by 0x55357FB: start_thread (pthread_create.c:465)
   by 0x5861B0E: clone (clone.S:95)
```

Figure 6.2: Error report from Endicheck run on the RadeonSI driver.

the check, we also needed to a add function for retrieving a mapping of an existing buffer, totaling about 40 added lines in a single place.

The GIT repository with the changes is in attachment 3 of this thesis. The repository includes the annotations and also the fixes described in the following sections.

Please note that the changes done to the `u_math.h` file in the attached GIT repository are more extensive than described above. This is because we have also added new byte-swapping functions, and introduced new types to avoid accidental mismatch between *concrete* endianess and *native* endianess integers. This brings the total of lines changed or added to this file to about 200 lines. Neither of these changes are necessary for annotating a program to be run under Endicheck and so were excluded in the analysis above.

6.1.3 Example Report

Figure 6.2 contains one of the error reports produced by the Endicheck tool. The tool was run on the `glxgears` demo program. The Mesa driver was annotated, as described in the previous section, and recompiled. All other components were unmodified. Endicheck was invoked with the following options:

```
--tool=endicheck --track-origins=yes
```

The error report itself can be divided into three main parts: the problem description, origin stack-trace and point-of-check stack-trace (in this order).

The problem description identifies the currently active thread, the nature of the problem and identification of the memory region containing the error. The memory region is identified by its address and an optional name provided by the analyzed program (in this case `radeon_add_buffer`). The metadata is printed for the part of the memory that contains data with the wrong endianess, using the following convention: `N` = native, `U` = undefined.

The first stack-trace is only present if origin tracking is enabled and is captured when the offending value is first created, for example by an arithmetic operation. It is possible that the data with wrong endianess come from multiple origins, in which case multiple stack-traces are printed.

We can see that the origin tracking stack-trace is very different from the second stack-trace. The second stack-trace is captured at the time when the endianess is checked, because annotated function is encountered. Both stack-traces help to pin-point the source of the error.

In this case, the error report has identified that an array of floating-point values[3] describing the *multisampling pattern* is not byte-swapped. We have to insert byte-swapping functions into `si_init_msaa_functions` where the floating-point array is produced. It is not possible to insert it into `si_set_constant_buffer`, since this function does not know the layout of the constant buffer and so cannot byte-swap it. It would also go against the Gallium API design, which states that the data passed into the Gallium driver should already be byte-swapped if necessary, by the upper gallium layers.

The fix is in the GIT repository in attachment 3, along with all the other fixes and changes we have done. If the fix is a result of an Endicheck report, we always include the original Endicheck report in the commit message, under the `ECNOTE` header.

6.1.4 Experiment

We wanted to verify that the error reports produced by Endicheck are comprehensive, that is to say that Endicheck will report all bugs present in the program. To test this, we have used the ability of our analysis to run even on architecture where endianess bugs do not occur. First of all, it is convenient to work on x86 based machines. But it will also ensure that all bugs found in this way are product of our analysis – since on x86 the endianess bugs do not manifest. If we would run Endicheck on PowerPC from the beginning, we would get extra help in

[3] IEEE754 floating point values also obey the endianess of the host platform, at least on x86, x64 and ARM.

finding those bugs, since they might manifest as crashes, images corruptions, etc. Running on x86 allows us to objectively attribute all found bugs to Endicheck.

First, we set a goal to fix the Radeon SI driver so that it can run the Glxgears demo on PowerPC, against a Radeon Sapphire Dual-X R7 265 2G D5 card (PCI ID 1002:6819). Without any fixes, Glxgears caused a GPU crash and subsequently crashed itself.

Initially, we have run Glxgears on x86 architecture and fixed all problems reported by Endicheck. After fixing all issues found by Endicheck, we moved the same graphics card to a PowerPC host and continued testing there. This allowed us to tell if the bug is detected by Endicheck or needs testing on real hardware to be discovered, as explained in the first paragraph.

In total, we needed to fix 6 bugs, out of which 4 were detected by Endicheck before testing on PowerPC. The remaining two bugs were not endianess related and thus were out of scope of our analysis. After the fixes, the Glxgears demo did successfully run. This shows that Endicheck successfully detected all bugs it was supposed to and provided reports useful enough so that the bugs could be fixed.

For the curious reader: the two other bugs were caused by different compiler behavior on the PowerPC platform. One of them was caused by the fact that the `char` type is signed on x86, while it is unsigned on PowerPC. The RadeonSI driver assumed it is signed. The other bug was caused by different layout of C structure bit–field, but the layout is not guaranteed by the GCC compiler or the C standard. We have rewritten the Radeon SI driver to use `uint32_t` instead and manipulate it using bitwise operations. This approach is platform independent.

6.2 Performance

Performance was not an important goal for Endicheck, as long as it is "reasonable" and the tool can be used for debugging of non-trivial programs. In this section, we compare performance of programs instrumented with Endicheck, programs instrumented with Memcheck and programs without any instrumentation.

The tables in this section will use following abbreviations:

- **MC**: Memcheck (`valgrind --tool=memcheck`)
- **EC**: Endicheck (`valgrind --tool=endicheck`)
- **-OT**: with origin tracking enabled (`--track-origins=yes`)
- **-IT**: with origin tracking enabled (`--track-origins=yes`), but imprecise (`--precise-origins=no`)
- **-P**: with memory protection enabled (`--protection=yes`)

6.2.1 Test Description

We use programs from the SPEC CPU2000 integer speed benchmark for measuring execution time. A newer SPEC CPU2017 is available, but is not suitable for our purpose. The programs in the new benchmark have memory requirements up

to 16 GiB and have longer run-times. When combined with the instrumentation overhead we are trying to measure, the memory and time requirements would be very high. Another reason is that the original [Nethercote and Seward, 2007] paper has used SPEC CPU2000, so our test will be somewhat comparable to the original Valgrind test.

The tests were run on a T550 ThinkPad notebook with 12 GiB of RAM and an i5-5200 processor clocked at 2.20 MhZ, under Arch Linux from circa Q2 2018. The SPEC2000 test harness was used to execute all tests, with iteration count set to three. Memcheck and Endicheck were both compiled from the same source, tagged as `thesis-benchmark` in attachment 1, with default options, by GCC v7.3.0. The benchmark named "gap" produced invalid results when compiled with this version of GCC and so was disqualified. One test also needed a small code fix to compile.

Our tests are divided into two runs, for performance reasons. The full range of configurations is run on the "test" data set provided by SPEC CPU2000. The results of this run are in Table 6.1. The test data-set is reduced compared to the full "reference" set and allows us to run the EC-OT analysis, which incurrs a huge slowdown, in a reasonable time. On the other hand, execution times for uninstrumented (native) runs are as low as 0.05 s. Benchmarking of these slow times might be imprecise and so we instead use MC as the base of comparison. Thus all times in the table are relative to MC, except native.

The second test run uses the full "reference" data set from SPEC CPU2000. These results, listed in Table 6.2 will be more precise and we use the native (uninstrumented) execution time as base of comparison.

6.2.2 Discussion of Results

As can be seen in Table 6.1, the average slowdown of Memcheck is 16.59x. This figure is close to the slowdown measured by [Nethercote and Seward, 2007], which is 22.1x. Endicheck, in comparison, slows down the analyzed program by factor of 35.31. This means Endicheck is roughly two times (2.13x) slower than Memcheck with default options.

According to table Table 6.2, the same relative slowdown is 1.64x. One of explanations for this difference is different ratio between the time spent instrumenting the code vs. time spent running the instrumented code in the long and short tests. The "perlbmk" test even shows a small speed-up in the short runs based on the test dataset. And even compared to the native execution time, the difference is small, leading to the conclusions that the test is bound by something else than processor performance. The full reference perlbmk test does not exhibit this behavior.

In general, the performance seems good, given that Endicheck is not as well optimized as Memcheck. Most users using Memcheck should be comfortable using Endicheck for their programs as well.

However, the performance of Endicheck with origin tracking is lacking compared to Memcheck with the same option. In practice, it was still usable for our Radeon SI OpenGL tests, but the measurement shows there should be a room for improvement.

Some of the relative slowdown between EC-OT and MC-OT might be un-

Program	Native (s)	MC (s)	MC-OT	EC	EC-P	EC-OT	EC-IT
bzip2	1.38	19.40	2.27x	2.07x	2.23x	33.87x	12.58x
crafty	0.70	18.70	2.21x	1.74x	1.78x	30.59x	11.07x
eon	0.09	6.60	1.73x	1.29x	1.34x	12.89x	4.23x
gap	—	—	—	—	—	—	—
gcc	0.31	12.70	1.96x	1.92x	1.98x	24.17x	9.53x
gzip	0.47	6.29	2.11x	1.86x	1.97x	41.97x	14.96x
mcf	0.05	0.85	2.38x	1.27x	1.32x	11.88x	7.08x
parser	0.66	10.50	2.19x	2.13x	2.28x	41.24x	16.29x
perlbmk	4.31	5.52	1.10x	0.95x	0.95x	1.17x	1.05x
twolf	0.05	1.64	1.88x	1.16x	1.20x	14.09x	5.51x
vortex	1.06	56.90	2.23x	1.95x	2.04x	28.38x	9.86x
vpr	0.49	8.02	2.00x	1.70x	1.75x	22.94x	8.30x
G.mean	0.41	7.86	1.97x	1.59x	1.65x	18.17x	7.56x

Table 6.1: SPEC CPU2000/test execution times, relative to Memcheck.

avoidable, because Endicheck must track origin information for much more data than Memcheck. From our tests, we conclude that creating the origin tag is the most expensive operation involved, probably because stacktraces must be collected and looked-up. When origin tags are created for each superblock, instead for each instruction, the run times drop roughly by a factor of two (see columns EC-OT and EC-IT).

Memcheck creates origin tags mostly only when allocating new unitialized memory and even in this case only one origin tag needs to be created for the memory block. In contrast, Endicheck creates origin tags for nearly every arithmetic operation.

6.2.3 Memory consumption

Another topic impacting the usability of Endicheck is the memory overhead it introduces. We will measure the Resident Set Size (RSS) of a demo program. If the demo program is run on computer with more memory than it needs, this is a good indicator of real memory requirements.

For the tests, we run unmodified Glxgears (no annotations) from Ubuntu Bionic under Valgrind. We record the memory map (`/proc/pid/smaps`) after Glxgears starts rendering frames and compute the RSS from the memory maps.

Please note that this value is different from what a typical system management tool, like KSysGuard, will show. These tools typically try to subtract memory shared by multiple processes. Memory overhead introduced by Valgrind tools is not shared, apart from the small constant size of Valgrind code itself.

The memory consumption of Endicheck, Memcheck, their variants and no instrumentation is shown in Table 6.3. Quite surprisingly, the memory usage of Endicheck is even lower than memory usage of Memcheck. Intuitively, the shadow memory should play major role in memory overhead, and Memcheck has better efficiency in this case. Valgrind documentation mentions about 1:4 shadow-to-normal memory ratio in practice, which Valgrind achieves thanks to shadow memory compression. Endicheck has 1:1 shadow memory ratio.

Program	Native (s)	MC	EC	EC-P
bzip2	66.3	11.63x	23.47x	24.45x
crafty	29.5	26.78x	48.10x	48.54x
eon	24.1	52.12x	93.36x	97.34x
gap	—	—	—	—
gcc	27.8	27.73x	116.62x	122.48x
gzip	79.9	8.92x	15.93x	16.80x
mcf	67.10	2.71x	6.90x	6.94x
parser	89.9	10.78x	23.04x	23.86x
perlbmk	45.9	38.45x	93.62x	96.27x
twolf	93	12.43x	19.77x	19.52x
vortex	43.8	44.36x	91.03x	92.85x
vpr	54.7	10.49x	20.29x	20.68x
G.mean	51.29	16.59x	35.31x	36.25x

Table 6.2: SPEC CPU2000/reference execution times and slowdown, relative to native runs.

Analysis	RSS (MiB)
Native	49.28
EC	190.00
MC	195.79
EC-PROT	190.03
EC-OT	589.21
EC-IT	396.16
MC-OT	308.72

Table 6.3: Memory usage of Glxgears running under different Valgrind tools.

However, Glxgears' RSS consists of 9.41 MiB of data and 39.88 MiB of code. Lot of the code does not need any shadow memory allocated, because it is never read. In practice, we have seen Endicheck allocate shadow memory only for 32.7 MiB of memory, which is only small part of the 190 MiB used, and other memory allocations, like instrumented code storage and debug information tables dominate.

Endicheck with origin tracking uses significantly more memory than Memcheck, which is mainly due to the larger number of origin tracking information stored. Valgrind's `--heap-profile=yes` option shows that Memcheck uses 3.5 MiB for the storage of origin tracking information, like stack-traces, while Endicheck uses 300 MiB. When the precision of origin tracking is reduced, the memory usage drops to 113 MiB.

Another reason for the higher memory usage with origin tracking enabled is the increased shadow memory ratio, from 1:1 to 3:1 – we now must allocate additional four bytes of the origin tag for each two bytes of memory. Memcheck stores the origin tags only for each four bytes, reducing the precision but also shadow memory usage.

6.3 Experience with Valgrind

For the benefit of other people planning to use Valgrind, we would like to discuss also our experience with using this instrumentation framework. In general, we were satisfied with it, but the documentation and multi-platform support is lacking.

The documentation is probably the major hurdle, since programmers documentation is virtually non-existing. The closest thing is the "Writing a New Valgrind Tool" and "The Design and Implementation of Valgrind" documentation sections. Since Valgrind is open-source, you can always check-out the source-code, of either the Valgrind core or Memcheck, which should help you get on track of writing a new tool. But a list of supported instruction and their meaning would be handy.

The next point is the not-so-perfect multi-platform nature of Vex IR. It abstracts the platform differences, but not perfectly, since only a subset of Vex IR instructions is implemented for a given platform. The subset is not documented, but depends on the needs of Memcheck and the platform's native instruction set. For example, the Vex IR on PowerPC does not support most operations on 64-bit integers and only supports comparison operations on 32-bit integers. These limitations are not part of the Valgrind design though, they probably result from lack of resources dedicated to supporting these architectures.

Originally, we also expected that Valgrind will provide helper functions to maintain the shadow memory infrastructure. Unfortunatelly, the shadow memory handling is hard-coded into Memcheck and not reusable. But re-implementing the shadow memory was not a significant challange.

7. Installation and Usage

To install the Endicheck tool, we recommend building it from the source, although binary packages for selected distributions are also available (attachment 2). If building from source, make sure that GCC, Make and Autotools are available. We have used Autoconf v2.69, Automake v1.15.1 and GCC v7.3, but it is highly likely that any reasonably recent version will work.

To build from a source package (attachment 1), run these commands:

```
cd valgrind
./autogen.sh
./configure
make
```

After running these commands, you have two options. One is to install Endicheck and run it from the installed location:

```
sudo make install
valgrind --tool=endicheck echo 'Testing Endicheck'
```

The second option is to run Endicheck directly from the source tree, without installation. This has to be done using a special script provided by Valgrind:

```
./vg-in-place --tool=endicheck echo 'Testing Endicheck'
```

The command-line options supported by Endicheck can be listed using the `--help` switch. Do not forget to combine `--help` with `--tool` to get help for the correct tool (Endicheck). If you encounter any problems during build or installation, consult the `README` files that come with Valgrind.

7.1 Annotations

The remaining part is analyzing your program. For that, you have to annotate your program. The annotations are available in the `valgrind/endicheck.h` headers. If Endicheck is not installed using `make install`, you have to specify the full path to the header.

The following annotations, called user requests in Valgrind terminology, are available:

- `EC_MARK_ENDIANITY(start, size, endianity)`
  Use this annotation in byte-swapping functions. The annotation marks a memory region from `start` to `start+size` as having endianity `endianity`. Most times, one wants to mark a memory region as `EC_TARGET`.

```c
#include <valgrind/endicheck.h>
#include <stdint.h>
#include <byteswap.h>
#include <unistd.h>
/* #include <endian.h> */

static uint32_t htobe32(uint32_t x) {
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
    x = bswap_32(x);
#endif
    EC_MARK_ENDIANITY(&x, sizeof(x), EC_TARGET);
    return x;
}

static int ec_write(int file, const void *buffer, size_t count) {
    EC_CHECK_ENDIANITY(buffer, count, NULL);
    return write(file, buffer, count);
}
#define write ec_write

int main() {
   uint32_t x = 0xDEADBEEF;
   x = htobe32(x);
   write(0, &x, sizeof(x));
   return 0;
}
```

Figure 7.1: Example program with Endicheck annotations.

- `EC_CHECK_ENDIANITY(start, size, msg)`
  Use this annotation in functions through which data leaves a program and that should be checked. The annotation checks that a memory region from `start` to `start+size` contains only data with *any* or *target* endianity. If the `--allow-unknown` option is passed to Endicheck, *unknown* endianity is also allowed.

- `EC_PROTECT_REGION(start, size)`
  Marks a region of memory from `start` to `start+size` as protected. Correct endianity is checked with each write to this region.

- `EC_UNPROTECT_REGION(start, size)`
  Un-marks a region of memory from `start` to `start+size` as protected.

- `EC_DUMP_MEM(start, size)`
  Dumps endianity of memory from `start` to `start+size`, even if it is correct. This is useful for debugging Endicheck itself.

A full working program demonstrating the two most important annotations (`MARK` and `CHECK`) is shown in Figure 7.1. If the `htobe32` call on line 23 is removed,

Endicheck will report a problem. The example also demonstrates ways to easily add annotations to standard functions, like `write` on line 19 and and `htobe32` on line 7.

8. Future Work

In this chapter, we describe some improvements that could be done to the Endicheck tool itself. We also present alternative approaches to detection of endianness bugs that we have not explored, but that might be viable.

Another work, not directly related to the analysis, is to make the Radeon SI driver fully usable and submit the patches to the upstream. To do so, we can run Endicheck on more complex programs and use the Piglit OpenGL test-suite [pig, n.d.]. In the current state, the RadeonSI driver is not yet usable and most programs other than Glxgears do not work yet[1].

8.1 Endicheck Improvements

This section list improvements that could be implemented in Endicheck right away without any architectural changes. They are:

- Test Endicheck with more programs, on more platforms and on more processor architectures supported by Valgrind. Currently we have only tested x86, x64 and 32-bit PowerPC on Linux.
- Report arithmetic instructions that use target endianness data. This would allow Endicheck to detect the other direction of endianess errors – programs using external data without byte-swapping.
- Provide replacement headers for `endian.h` from the C library. These replacement headers would remove the need for annotations in programs using the standard byte-swapping functions. Similarly, provide replacement for `arpa/inet.h`.
- Automatically check syscalls such as `write` for correct endianness.
- Read debugging information to correctly guess `.data` memory type. This could minimize false positives caused by unrecognized byte-sized data.
- Add automatic byte-swap detection based on static analysis of executed code.
- Support suppression files to suppress eliminate positives. Valgrind already has helper functions for this and Memcheck supports suppressions.
- Have multiple origin tags for each 16-byte unit of large types stored in registers, instead of single tag.

[1] Actually it is somewhat amusing to watch the Glxgears run on PowerPC for the first time, while the window manager decorations are randomly thrown around the screen and corrupted.

8.2 Porting to Higher-level Languages

Our analysis was designed to work mainly with C-like languages. The rationale for this decision was discussed in Section 1.1.

Endicheck was tested only with programs written in C/C++ and is not prepared to work directly with other C-like languages (Ada, D, Rust, etc.). We use the Valgrind user request mechanism for annotations, which is provided in the form of C macros. These macros expand to special inline assembly sequences that Valgrind can recognize.

If the user wishes to use a different language than C, the user-request mechanism would need to be ported to another language. The other option is to wrap the user requests as regular C functions, for which bindings can be created in most languages. This could be also done independently by the user, without changing the Endicheck core.

Technically it is possible to run high-level managed languages like Java or C# (via Mono) under Valgrind. However this is mostly done to analyze the lower-level components that the higher-level languages rely on (native C libraries, etc.), not the managed program itself. But it still could be possible to use the same analysis for the managed program, however this was not tested – we must be prepared for unforseen ~~consequences~~ interactions with garbage collector and other runtime components.

An analysis tailored to a managed programming language can be faster, more precise and easier to use, since it can make more assumptions about the analyzed program. However, if an analysis works on byte-code level, it cannot track metadata using the shadow memory technique described in Section 5.3. For example, the Java byte-code does not allow to determine a memory address of a variable. Even if it was possible, the garbage collector can rearrange objects in memory and change the address at any time.

Instead, a different shadow memory technique can be adopted, which modifies the structure definitions (class definition in Java) to directly hold the shadow data. A good and more detailed example of this shadow memory technique is given in [Bell and Kaiser, 2014], where it is used for taint analysis of Java programs.

However, languages that allow this kind of structure reorganization are precisely the languages that would benefit from our analysis the least. If a language allows the structure layout to be changed, it means that the byte-code must be fully isolated from the structure layout. This means it must be prevented from reading the memory where the structure is stored as a plain array of bytes. This also effectively prevents it from knowing the endianness of the structure members and having any kind of endianness errors.

.NET and thus C# sits somewhere in the middle between Java and lower-level languages. On one hand, it is byte-code based, but on the other hand it offers `unsafe` blocks, explicit structure layout, pointers and P/Invoke calls to native code[2]. We are not aware of any .NET instrumentation using shadow memory (such as taint analysis) and we are not sure what approach would be ideal for C#.

[2]Java has JNI, but JNI prevents the native code from accessing the Java objects directly

8.3 Alternative: Probabilistic Data Analysis

It would be interesting to investigate to which degree an endianess can be determined simply from the data. For example, a sequence of bytes `0x1`; `0x0`; `0x0`; `0x0` can either be a little-endian integer 1 or big-endian integer 2^{31}, however intuitively, it will probably be the smaller integer. But we are not aware of any research into average distribution of numbers inside computer programs that would prove this intuition.

But a general principle, called *Benford's law*, observes that random integers from most probability distributions have specific distribution of leading digits. Since the leading digit changes based on the endianess used to interpret the number (unless we choose base 256), the law could be used to establish probability for the endianess.

The analysis is complicated by the fact that unless the programmer describes the structure of the data, we do not know where the integers start and end. However, the analysis might rely on further assumptions, like alignment requirements, to overcome this problem.

8.4 Alternative: Comparative Runs

A classic way to test for endianess bugs is to run program on architecture where the program will have to do byte-swapping. Commonly this is done with help of virtualization, for example using QEMU. This does not guarantee that the errors would be caught, they can just cause silent data corruption.

To make this easier, we could run the a program on two machines, once on a big-endian architecture and once on little-endian architecture and compare the data leaving both variants of the program. The main grunt of work would probably be to ensure that both programs run in lockstep and the whole process is user-friendly. However, it is possible that this problem is already solved by some existing framework or library for other testing purposes.

The testing process could be done either on separate computers, using virtualization or using a special compiler, simulating an architecture of different endianess (this so-called biendian compiler for x86 is described by [Domeika, 2011]).

If the implementation issues are solved correctly, this approach has the potential to be the most accurate. It can even detect problems if the data leaving the program are encrypted or compressed (but the precise localization of mismatched data will naturally not be possible in this case).

However, the approach cannot always detect when the program forgets to byte-swap data entering the program, unless the error affects one of the outputs with *concrete* endianess. For example, a program that reads big-endian data and produces native-endianess output cannot be checked, since the native endianess output will always be different between little-endian and big-endian runs of the program.

Conclusion

We have developed a new dynamic analysis tool, Endicheck, which provides a useful alternative to tools like Sparse for detecting endianess-related bugs. One of our main goals was to provide a tool that can be used in different – and easier – way than the existing static analysis tools based on variable annotation.

We have succeeded in that. Endicheck is able to detect missing byte-swaps in analyzed program mostly automatically, and only few source code changes must be done by a programmer. Future work might improve this even further, by automatically detecting byte-swapping functions.

Endicheck was tested on a Radeon SI OpenGL driver, which has proved it (1) can handle large complex programs, (2) does not require lot of source code changes and (3) can identify endianess bugs. We evaluated the Endicheck tool on a single test OpenGL program (`glxgears`), for which Endicheck correctly identified all four endianess-related bugs before we tested the program on a big-endian hardware.

Our subjective experience was that the provided error reports were helpful and allowed us to pin-point the bug locations quickly, even though we were unfamiliar with the code base. The reports are improved by the origin tracking feature, borrowed from Memcheck, which helps to find the causes of the error reports.

Endicheck could also be used to in automatic testing scenarios, as an alternative to testing the program on both little and big endian architecture. A testing environment based on Endicheck might be easier to set-up and maintain than environment based, for example, on virtual machines.

Bibliography

[JPF, n.d.] Jpf wiki. URL https://github.com/javapathfinder/jpf-core/wiki. Accessed 30.4.2018.

[cla, n.d.] Clang 7 documentation / memorysanitizer. online. URL https://clang.llvm.org/docs/MemorySanitizer.htmlhttps://clang.llvm.org/docs/MemorySanitizer.html. Accessed 3.3.2018.

[dat, n.d.] Clang 7 documentation / dataflowsanitizer. online. URL https://clang.llvm.org/docs/DataFlowSanitizer.html. Accessed 3.3.2018.

[dyn, n.d.] Dynamorio website. online. URL http://www.dynamorio.org/home.html. Accessed 30.4.2018.

[ker, n.d.] Building the kernel with clang. URL https://lwn.net/Articles/734071/. Accessed 30.4.2018.

[pig, n.d.] Piglit. online. URL https://piglit.freedesktop.org/. Accessed 24.4.2018.

[spa, n.d.] Sparse documentation. Linux GIT. URL https://www.kernel.org/doc/html/latest/dev-tools/sparse.html. v4.15.

[Bell and Kaiser, 2014] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 83–101, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660212. URL http://doi.acm.org/10.1145/2660193.2660212.

[Bond et al., 2007] Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley. Tracking bad apples: Reporting the origin of null and undefined value errors. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2007)*, Montreal, Canada, October 2007. URL http://valgrind.org/docs/origin-tracking2007.pdf.

[Brown, 2016] Neil Brown. Sparse: a look under the hood. online, 2016. URL https://lwn.net/Articles/689907/. Accessed 30.4.2018.

[Bungale and Luk, 2007] Prashanth P. Bungale and Chi-Keung Luk. Pinos: A programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 137–147, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1. doi: 10.1145/1254810.1254830. URL http://doi.acm.org/10.1145/1254810.1254830.

[Burrows et al., 2003] Michael Burrows, Stephen N. Freund, and Janet L. Wiener. Run-time type checking for binary programs. In Görel Hedin, editor, *Compiler Construction*, pages 90–105, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36579-2.

[Collingbourne, 2013] Peter Collingbourne. Datalow sanitizer design discussion. llvm-dev mailing list, 2013. URL https://groups.google.com/d/msg/llvm-dev/xz2jkpBmFbY/uxiNUHYeYtYJ.

[Corbet, 2006] Jonathan Corbet. Using sparse for endianness verification, 2006. URL https://lwn.net/Articles/205624/. Accessed 5.2.2018.

[Domeika, 2011] Max J. Domeika. Bi-endian compiler: A robust and high performance approach for migrating byte order sensitive applications. In *Proceedings of ESA'11 The 2011 International Conference on Embedded Systems and Applications*, 2011.

[Luk et al., 2005] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[MinGW team, n.d.] MinGW team. Mingw-w64 - contribute. URL https://mingw-w64.org/doku.php/contribute. Accessed 8.4.2017.

[Nethercote and Seward, 2007] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, San Diego, California, USA, June 2007. URL http://valgrind.org/docs/valgrind2007.pdf.

[Newsome and Song, 2005] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, 2005.

[Nicholas and Seward, 2007] Nicholas 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. URL http://valgrind.org/docs/shadow-memory2007.pdf.

[Oracle, 2017] Oracle. Java® platform, standard edition & java development kit version 9 api specification, 2017. URL https://docs.oracle.com/javase/9/docs/api/overview-summary.html. Accessed 31.3.2017.

[Schwartz et al., 2010] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy*, pages 317–331, May 2010. doi: 10.1109/SP.2010.26.

[Seward and Nethercote, 2005] Julian Seward and Nicholas 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. URL http://valgrind.org/docs/memcheck2005.pdf.

[Song et al., 2008] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89861-0. doi: 10.1007/978-3-540-89862-7_1. URL http://dx.doi.org/10.1007/978-3-540-89862-7_1.

List of Figures

List of Tables

Attachments

1. **Endicheck source code**

   Source code of valgrind, with endicheck modifications and GIT history. Version distributed with the thesis is tagged `thesis`.

Online:	https://github.com/rkapl/endicheck
Thesis attachment:	`endicheck.zip`

2. **Endicheck binaries**

   Binary packages for selected distributions. Version available online might be newer than the one distributed with thesis.

Online:	https://build.opensuse.org/project/show/home:rkapl:endicheck
Thesis attachment:	`endicheck-dist.zip`

3. **Mesa source code**

   Source code of Mesa, with Radeon SI endianess fixes and GIT history. Version distributed with the thesis is tagged `thesis`.

Online:	https://rkapl.cz/repos/git/roman/mesa
Thesis attachment:	`mesa.zip`

Thesis source version identifier: 5fb7faf