# Improving Driver Robustness: an Evaluation of the Devil Approach

Laurent Réveillère, Gilles Muller

# Improving Driver Robustness: an Evaluation of the Devil Approach

Laurent Réveillère
LaBRI, University of Bordeaux I
F-33405 Talence Cedex, France

Gilles Muller
IRISA/INRIA
F-35042 Rennes Cedex, France

E-mail: Laurent.Reveillere@labri.fr
Gilles.Muller@irisa.fr
Web: http://compose.labri.fr

## Abstract

*To keep up with the frantic pace at which devices come out, drivers need to be quickly developed, debugged and tested. We have recently introduced a new approach to improve driver robustness based on an Interface Definition Language, named Devil. Devil allows a high-level definition of the communication of a device. A compiler automatically checks the consistency of a Devil specification and generates stubs that include run-time checks.*

*In this paper, we use mutation analysis to evaluate the improvement in driver robustness offered by Devil. To do so, we have injected programming errors using mutation analyses into Devil based Linux drivers and the original C drivers. We assess how early errors can be caught in the development process, by measuring whether errors are detected either at compile time or at run time. The results of our experiments on the IDE Linux disk driver show that nearly 3 times more errors are detected in the Devil driver than in the original C driver.*

## 1. Introduction

A device driver is a key system component that connects the device and the operating system kernel. Device drivers are critical both in general-purpose computers and in the fast-evolving domain of appliances; in these areas new hardware devices are introduced at a frantic pace. If driver development takes too long or the software robustness is too low, such a hardware innovation may turn into a disaster.

It is well known in the operating system community that device drivers are low-level code and thus error-prone. This observation is particularly relevant to *hardware operating code*, (i.e., the layer in charge of communication with the hardware). Hardware operating code consists of assembly-level operations, even if written in C. Therefore, it is obviously prone to errors and requires tedious debugging. There are three common sources of programming errors in the hardware operating code: (i) an incorrect understanding of the device interface, (ii) a typo in the assembly-level code, (iii) a reuse, in an inappropriate context, of an optimized sequence of accesses to the device. These errors proliferate, because there are no tools to help develop such code or to verify its correctness.

Recently, we introduced a new approach towards improving driver robustness, based on a Interface Definition Language (IDL) named Devil [12]. Devil is aimed at specifying the communication layer with a device, providing a typed, functional interface. This approach shifts the assembly-level code into stub functions that are generated automatically by the Devil compiler from a device specification (see Figure 1). Using Devil improves driver robustness by addressing common sources of errors in hardware operating code:

- Devil specifications can be checked for consistency by the Devil compiler. This helps the device expert in writing a correct specification. This also offers a degree of confidence to the driver programmer who uses an existing specification to develop a driver.

- A Devil specification clearly documents a device interface and serves as a knowledge repository. The driver programmer no longer as to rely on often imprecise or inaccurate vendor documentation. In fact, in our vision, Devil specifications either should be written by device vendors or should be available as public libraries.

- Thanks to automatic stub generation, the driver programmer no longer has to write low-level error-prone code.

- Correct usage of generated stubs by the driver programmer can be verified at compile time and at run time.

Devil has been proved to be expressive enough to specify a wide variety of devices such as DMA, interrupt, Ethernet, IDE disk, sound, mouse and video controllers. Additionally, we have shown that Linux drivers re-engineered using Devil are almost as efficient as the original ones [11]. These experiments demonstrate the interest of the approach and the ability to develop real drivers. In this paper, we perform a quantitative evaluation of the benefits of Devil in terms of safety and robustness.

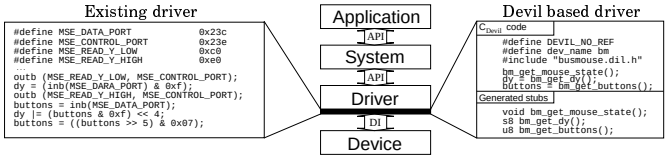


**Figure 1. Developing drivers with Devil**

## This paper

Evaluating an improvement in robustness is a complex task for which there is no simple or well-established method. In the case of Devil, we need to measure how early programming errors can be caught in the development process. To do so, we have to simulate errors, and inject them into both traditional C drivers and Devil ones. Finally, we need to measure the error detection coverage of the various mechanisms implemented in each language to catch errors at specification time, at compile time and at run time.

In this paper, we are mainly interested in evaluating the detection of typographical errors. Errors resulting from an incorrect understanding of the device or an inappropriate optimization are covered by the Devil approach itself. In fact, our experience in re-engineering existing drivers using Devil shows that the hard part is to write specification that actually match the real behavior of devices. As such, Devil specifications are the repository of the device expert knowledge. By using existing Devil specifications, the programmer is guided through the driver implementation process.

Typographical or inattention errors are highly problematic because they can remain hidden for a long time, and because they do not always result in an instantaneous system crash. There is significant potential for such errors in a language like C. For instance, the hexa-decimal notation makes hard to detect additional or missing characters; `0xfffff` looks similar to `0xffffff`. Inattention errors often result from the use of a value defined in another part of the program; for example confusion in register names is quite frequent. Finally, optimization errors mostly occur when code is copied without checking the validity of the execution context.

To catch typographical errors at development time, Devil allows specific debug stubs to be generated from a device specification. Debug stub interfaces are designed so that an incorrect usage of the stub is likely to raise a type error at compile time. Also, debug stubs systematically contain run-time assertions[1] that verify additional properties on usage parameters and check that the device behavior actually matches it specifications.

To measure the coverage of our mechanisms, we have defined two error models. One is for typos in Devil specifications, the other one is for typos in the C language. For each model, a dedicated set of rules describes mutation operators which are used to generate mutants. Specification mutants are checked by the Devil compiler which detects inconsistencies in the specification. C driver mutants are compiled using the standard C compiler. If no error is reported at compile-time, the mutant driver is then run within the Linux kernel. The result of the run (i.e., assertion raised, system crash, no-failure) is then reported.

We have applied C mutations to both original Linux drivers and the equivalent Devil re-engineered drivers. For original Linux drivers, mutations are applied only in the hardware operating code. For Devil drivers, mutations are applied at the call sites of the generated stubs. By compiling and running mutants, we are able to measure how many errors can be detected at each stage of the development process. The latter result allows us to evaluate the robustness improvement induced by using Devil in driver development.

To summarize, our contributions are the following:

- We describe how to generate debug stubs that improve detection of programming errors when using Devil in developing drivers.
- We have modeled typographical errors using a set of rules which are then used to defined mutation operators. Two sets of mutation operators, one for the C language and one for Devil have been defined.
- We have defined a methodology for evaluating the robustness improvement induced by Devil. For that, we compile and run drivers which have been mutated so as to introduce programming errors.
- We have applied our test methodology to the IDE Linux disk driver by comparing the original Linux driver with a Devil re-engineered one. Our results show that 72% of the errors in the Devil driver are detected either at compile time or at run time. This

[1] Run-time assertions are often used during development for testing purpose.

represents nearly 3 times more errors than are detected in the original C driver.

- Finally, we have injected errors into several Devil specifications. Up to 90% of the errors were detected by the Devil compiler.

The rest of this paper is organized as follows. Section 2 briefly presents the Devil language, and describes specification verification and the generation of stubs in debug mode. Section 3.3 presents mutation rules introduced to simulate programming errors. Section 4 describes the methodology for measuring the robustness of C drivers as compared to Devil based drivers and details the results of our experiments. Section 5 describes related work. Section 6 concludes and suggests future work.

## 2. Robust Driver Development with Devil

Devil is an IDL aimed at describing the functional interface of a device. From a Devil specification, a compiler generates stubs containing low-level code to operate the device. The development of a driver using Devil is a two-phase process: (i) generation of stubs for the specific hardware/software context in which the device is used, (ii) development of code to glue the upper layers of the driver to the generated stubs. This glue code consists of procedure calls written in C; we name this glue code $C_{Devil}$ in the rest of the paper.

In the rest of this section, we first present an overview of the Devil language. Then, we describe how Devil specifications can be checked so as to detect possible inconsistencies. Finally, we show how to generate stubs and headers that allows typographical errors in $C_{Devil}$ to be detected both at compile time and at run-time.

### 2.1 Overview of the Devil language

To design Devil, we have studied a wide spectrum of devices and their corresponding drivers, mainly from Linux sources: Ethernet, video, sound, disk, interrupt, DMA and mouse controllers. This study was supported by literature about driver development [4, 13], device documentation available on the web, and discussions with device driver experts for Windows, Linux and embedded operating systems.

Concretely, a device can be described by three layers of abstraction: *ports*, *registers*, and *device variables*. The entry point of a Devil specification is the declaration of a device, parameterized by *ports* or ranges of ports, which abstract physical addresses. Ports then allow device *registers* to be declared; these define the granularity of interactions with the device. Finally, *device variables* are defined from registers, forming the functional interface to the device. Figure 2 illustrates a schematic example of the relationship between the various levels of abstraction.

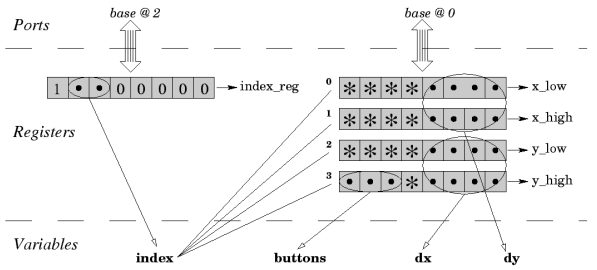


**Figure 2. Schematic view of a fragment of the Logitech Busmouse specification**

We now define ports, registers, and device variables by describing the Devil specification of the Logitech Busmouse (see Figure 3).

**Ports.** Depending on how a device is mapped, it can be operated via either I/O or memory operations. To hide this complexity, we define ports, which are at the basis of the communication with the device. A device often has several communication points whose addresses are derived from one or more base addresses. Therefore, the port constructor, denoted by `@`, takes as arguments a ranged port and a constant offset (*e.g.*, `base@1` as illustrated by line 4 of the Busmouse specification). To enable verification, the range of valid offsets must be specified within the entry point declaration (*e.g.*, `port@{0..3}` as illustrated by line 1 of the Busmouse specification).

**Registers.** Registers define the granularity of interaction with a device; as such the register size (in number of bits) must be explicitly specified. Registers are typically defined using two ports: one for reading and one for writing. Only one port needs to be provided when reading and writing share the same port, or when the register is read-only or write-only.

A register declaration may be associated with a mask that specifies bit constraints. An element of this mask can be '`.`' to denote a relevant bit, or '`0`' or '`1`' to denote a bit that is irrelevant when read but has a fixed value (`0` or `1`) when written, or '`*`' to denote a bit that is irrelevant when read or written. As an example, consider the declaration of the `write-only` register `index_reg` in line 16 of the Busmouse specification.

```
register index_reg = write base@2,
                     mask '1..00000' : bit[8];
```

```
device logitech_busmouse (base : bit[8] port @ {0..3})                                  1
{
   // Signature register (SR)
   register sig_reg = base @ 1 : bit[8];                                                4
   variable signature = sig_reg, volatile, write trigger : int(8);                      5

   // Configuration register (CR)
   register cr = write base @ 3, mask '1001000.' : bit[8];                              8
   variable config = cr[0] : { CONFIGURATION => '1', DEFAULT_MODE => '0' };             9

   // Interrupt register
   register interrupt_reg = write base @ 2, mask '000.0000' : bit[8];                  12
   variable interrupt = interrupt_reg[4] : { ENABLE => '0', DISABLE => '1' };          13

   // Index register
   register index_reg = write base @ 2, mask '1..00000' : bit[8];                      16
   private variable index = index_reg[6..5] : int(2);                                  17

   register x_low  = read base @ 0, pre {index = 0}, mask '****....' : bit[8];         19
   register x_high = read base @ 0, pre {index = 1}, mask '****....' : bit[8];         20
   register y_low  = read base @ 0, pre {index = 2}, mask '****....' : bit[8];         21
   register y_high = read base @ 0, pre {index = 3}, mask '...*....' : bit[8];         22

   variable dx = x_high[3..0] # x_low[3..0], volatile : signed int(8);                  24
   variable dy = y_high[3..0] # y_low[3..0], volatile : signed int(8);                  25
   variable buttons = y_high[7..5], volatile : int(3);                                 26
}
```

**Figure 3. Specification of the Logitech busmouse**

This mask indicates that only bits 6 and 5 are relevant. Also, bit 7 is forced to `1` when written while bits 4 through 0 are forced to `0`. Proper register masking is performed by the stubs generated by the Devil compiler.

**Device variables.** In order to minimize the number of I/O operations required for communicating with a device, hardware designers often group several independent values into a single register. Accessing these values requires bit mask and shift operations, which are error-prone in a general programming language such as C. Devil abstracts values as device variables, which are defined as a sequence of bit registers. Device variables are strongly typed in order to detect potential misuses of the device. Possible types are booleans, enumerated types, signed or unsigned integers of various sizes, and ranges or sets of integers. In line 17 of the Busmouse specification, the 5th and 6th bit of the `index_reg` register make up a two-bit unsigned integer variable (*i.e.*, a variable that can take a value from 0 to 3). The `private` attribute means that the `index` variable is not defined in the functional interface of the Busmouse controller and thus can not be directly accessed by the driver programmer.

**Access pre-actions.** Device functionalities are often extended by mapping multiple registers to a single physical address. Examples are index-based addressing mode and banks of registers. As a result, accessing such registers requires the setting of a specific context which may involve several I/O operations. To capture this situation, Devil allows pre-actions to be attached to a register. For example, lines 19 and 20 of the Busmouse specification declare two read-only registers on the same port `base@0`, provided that the variable `index` is set either to `0` or `1` prior to the port access.

```
register x_low  = read base@0,
                  pre {index = 0},
                  mask '****....' : bit[8];
register x_high = read base@0,
                  pre {index = 1},
                  mask '****....' : bit[8];
```

**Register concatenation.** Device variables can be spread over several registers. As illustrated by line 24 of the Busmouse specification, constructing the `dx` variable requires concatenation of the two registers `x_high` and `x_low`. The 8-bit variable `dx` is obtained by concatenating the four lower bits of register `x_high` with the four lower bits of register `x_low`.

```
variable dx = x_high[3..0] # x_low[3..0], ...
```

Many features of Devil are not detailed here. These features include enumerated types, facilities for specifying contorted addressing modes, and complex register and variable declarations. A complete description of the language can be found in [14].

## 2.2 Specification correctness

Devil is both a strongly typed language and a layered language. The layering of abstractions introduces redundancy in the information provided in a specification and thus opens

```
/* Type representation */
struct Drive_t_ {const char *filename; int type; u32 val; };
typedef struct Drive_t_ Drive_t;
static const Drive_t MASTER = {__FILE__,4,0x0u};
static const Drive_t SLAVE  = {__FILE__,4,0x1u};

/* write stub for the ide_select register  */
static inline void reg_set_ide_select (u8 v) {
  outb ((u8)v, base + 6);
  cache.cache_ide_select = v;
}

/* read stub for the ide_select register*/
static inline u8 reg_get_ide_select () {
  return inb (base + 6);
}

/* write stub for the Drive variable */
static inline void set_Drive (Drive_t v) {
  u8 tmp_0 = cache.cache_ide_select & 0xefu | ((u8)v.val & 0x1u) << 4;
  reg_set_ide_select (tmp_0);
}

/* read stub for the Drive variable */
static inline Drive_t get_Drive () {
  Drive_t v;
  u8 tmp_v = (reg_get_ide_select () & 0x10u) >> 4;
  v.filename = __FILE__; v.type = 4; v.val = (u32)tmp_v;
  return v;
}
```

**Figure 4. Debug stub for the IDE Drive variable**

opportunities for verifications. Nevertheless, the Devil syntax ensures that information is always introduced in a layer-specific manner. Therefore, there is no text redundancy in a specification. Consistency properties can be verified both within abstraction layers, and across abstraction layers.

**Intra-layer consistency**

Within a layer, we can verify type properties and the uniqueness of entities:

- All uses of Devil abstractions (*e.g.*, ports, registers, variables) can be matched against their definition to check type correctness. The type of a register or variable describes usage constraints, such as whether the register or variable is read or write only. Also, various size checks can be performed: the size of data accesses on ports, the size of registers, the size of variables derived from conversion functions, the size of bit masks, and the size of bit patterns that are associated a symbolic name in enumerated types, port ranges, and bit ranges for register fragments.

- The following must be uniquely defined: port parameters in a device declaration, ports, registers, types, symbolic names and bit patterns in enumerated types and variables.

**Inter-layer consistency**

The layered structure of Devil implies that declarations of entities at each level of abstraction have to be consistent with the corresponding declarations at lower layers: variables are defined from registers and a register from ports. For example, the attribute (read, write, or read/write) of a variable must be consistent with the attributes of the registers from which the variable is defined. Inter-layer consistency also implies that all entities defined at a given layer must be used in the next level. Finally, Devil entities such as ports and registers must not overlap.

- The no omission constraint concerns port arguments in a device declaration, values of ranged port offsets, registers, and register bits. Read elements of a type mapping must be exhaustive. Also, a type for reading (as well as possibly writing) must be used with a readable variable. The same holds for writing.

- Each port must appear only once in the register definitions, except when registers are defined using disjoint pre-actions or masks. However, a single port may be used for reading by one register and writing to another. No bit of a single register can be used in the definition of two different variables.

## 2.3 Generation of debug stubs

Stubs are generated from a device specification by the Devil compiler. The implementation of a stub depends on

whether the driver is compiled in production (ie performance) mode or in debug mode. In production mode, stubs are implemented so as to be as efficient as possible. The generated C code is low level and only limited verification can be done by the compiler when compiling the $C_{Devil}$ layer.

In debug mode, the stub code and header have been designed so that a typographical error in $C_{Devil}$ is likely to raise an error at compile time. Also, stubs contain run-time assertions that check parameter bounds. Finally, assertions are also generated after a variable read so as to verify that the read value matches the variable type. These assertions verify that a device behaves accordingly to its Devil specification.

#### Compile-time type verifications

Detecting a typographical error at compile time implies that the C compiler is able to raise a type error. This means that representation of Devil types should be as precise as possible. In other words, each Devil type should be represented by a new C type. Because the C compiler only raises type errors for incorrectly-used structures, we encode each Devil type as a distinct C structure type.

As an example, consider the following specification of the `Drive` variable in an IDE disk controller. This variable has an implicit type represented by a enumeration of the values SLAVE and MASTER.

```
register ide_select = base@6, mask '1.1.....';
...
variable Drive = ide_select[4]: {
  SLAVE  <=> '1',
  MASTER <=> '0'
};
```

Figure 4 presents stubs generated for the IDE `Drive` variable. As shown in lines 1-5, a specific C structure `Drive_t_` is generated to represent the Devil type of `Drive`. Two constant instances of `Drive_t_` are created to represent the values MASTER and SLAVE.

Representing Devil types as structures implies that the comparison operator of C is no longer available. Since operator overloading is not available in C, a possible solution to implement equality testing would be to provide a specific function for each Devil type. While such a solution preserves compile-time verification of types, it introduces additional complexity in the development and the readability of the $C_{Devil}$ code. To keep simple the writing of the $C_{Devil}$ code, we chose to have a single function (implemented as a macro) for comparing variables independently of their type. However, this approach implies that type verification is performed at run time.

#### Run-time assertions

To verify the type of a Devil variable at run-time, we have to provide a unique type identifier and to store it in the representation of the Devil variable. This is done by the `filename`, and `type` fields of the `Drive_t_` structure. `filename` is filled by the compiler using the `__FILE__` ISO C macro. `type` is a specification-unique counter that is generated by the Devil compiler. As an example, `type` is equal to 4 in the case of the type of the `Drive` variable. When reading a variable `filename` and `type` are initialized as shown in line 28 of Figure 4. Using the informations stored in `Drive_t_`, the following macro `dil_eq` performs a run-time type check and raises a error message in case of a type mismatch.

```
#define dil_assert(expr) ((expr) ? 0 : \
  panic("Devil assertion failed in file %s \
         line %d", __FILE__, __LINE__))

#define dil_eq(x, y) ( \
  dil_assert(!strcmp(x.filename, y.filename) \
             && x.type==y.type), \
  x.val==y.val)
```

Additional assertions are generated for types that are specified as a fixed set of integers, so as to check that the value is in the right range. For example, the stub for reading a variable of type `int{0,2,3}` contains an assertion that verifies that the value read is a two-bit integer that is not equal to 1.

Finally, run-time assertions also enable additional verifications related to the correctness of the specification of a device. For example, consider a register that has a fixed value for a given bit. If the value read at run time from the register does not respect the mask specification, then either the specification is incorrect, or the device does not behave correctly.

## 3. Mutation Analysis

Mutation analysis is a fault-based testing technique for unit-level testing. This technique was introduced twenty years ago [3, 6] and it has been successfully used to test C, Fortran and ADA programs [5, 9]. More recently, mutation analysis has enabled automatic test-data generation [7].

For a program $P$, mutation testing produces a set of alternate programs. Each alternate program, $P_i$, known as a *mutant* of $P$, is obtained by modifying one statement of $P$ at a time, according to *mutation rules*. These mutation rules are derived from studies of errors commonly made by programmers when translating requirements into code [2].

In traditional mutation testing, one wants to evaluate the coverage of a set $t$ of tests with respect to a program $P$. The principle of mutation testing is that if $t$ adequately covers $P$, then at least one of the tests in $t$ should be able to discriminate $P$ from a mutant $P'$. The proportion of mutants that *die* during mutation testing indicates how well $P$ is covered by $t$.

In our study, we are interested in measuring the coverage of the compile-time analyses and run-time assertions. Therefore, we consider the analyses and assertions to be the set of tests $t$. The tests $t$ adequately cover a program $P$ if an analysis or assertion can discriminate $P$ from every mutant $P'$.

The Devil approach clearly stages driver development into two phases, device specification in Devil and C code development. Therefore, we evaluate the coverage of programming errors at both stages. To do so, we have designed two sets of mutations rules, one for Devil and the other for C.

### 3.1 Error model

Our purpose is to model typographic and inattention errors. Ideally, the model should be designed by studying errors made by industry driver developers. However, to our knowledge, such data is not available. Therefore, we designed mutation rules from our past experience in designing drivers and by studying bug reports from Linux drivers.

Typographical errors are the result of an additional character, a missing character or a replaced character in a literal constant. For example, given a 2-digit base-10 number, 50 mutants can be generated: 2 for removing a digit, 30 for inserting a new digit, and 18 for replacing a digit. Additionally, typographical errors can be the result of the choice of an erroneous operator. Mutation rules for both kinds of errors ensure that character or operator replacement is always performed within the equivalent class of symbols.

Inattention errors often come from the confusion of names of identifiers that have similar purposes. We reproduce such errors by replacing an identifier by another one defined in the same file. The identifier is chosen from among the identifier declared at a same level of abstraction as the mutated identifier.

Finally, mutation rules are always defined such that mutants are syntactically correct, and have a different semantics than the original program. As a result, mutants cannot be detected by a simple syntax analysis. Their detection requires checking type properties.

### 3.2 Mutation rules for Devil

Because Devil is a language dedicated to the specification of device interfaces, it does not have as many constructs as a general purpose language like C. This obviously reduces the number of possible mutations.

- mutations on literals. Devil literals are either decimal or hexadecimal constants, or domain specific values such as bit strings (`0`, `1`, and `*`), and bit patterns (`0`, `1`, `*` and `.`). Character replacement or insertion is always performed within the same semantic class (*e.g.*, bit string, decimal or hexadecimal value).

- mutations on operators. Operators are used to specify integer ranges ("`,`" and "`..`") and type definitions ("`<=`", "`=>`" and "`<=>`"). Mutants are chosen within the same semantic class.

- mutations on identifiers. Identifiers are names of registers, variables and types. Mutations for identifiers are always performed within the same semantic class (register, variable or type). Also, no mutation is introduced at the declaration site of the name of a Devil variable (i.e., left side of the "=" character). Such a mutation would only affect the stub name rather than change the semantics of the specification.

### 3.3. Mutation rules for C

In a C driver, we are only interested in testing the hardware operating code. Thus, we manually insert tags to mark the corresponding regions in the original C driver and only mutated the tagged regions. In a Devil-based driver, mutations are only applied to the $C_{Devil}$ code.

- mutations on literals. C literal are the usual decimal, octal or hexa-decimal values. Mutations follow the rules defined in section 3.1.

- mutation on operators. Mutations on operators reflect common habits in writing hardware operating code. First, hardware operating code contains many bit manipulations. Second, in C values that are not structures are always considered by the compiler as integers. Finally, expressing a bit mask is commonly done by using the binary operator '|', but some programmers prefer the operator '+' which possesses a different semantics. Mutation rules are summarized in Table 1.

| operator | mutant | operator | mutant | operator | mutant |
|---|---|---|---|---|---|
| & | && \| | + | \| & | \| | & + |
| << | >> < | >> | << > | ~ | ! |

**Table 1. Mutation rules for C operators**

- mutation on identifiers. Driver programmers commonly define macros for ports, indexed registers, bit patterns, etc. Even if these macros refer to distinct abstract entities (i.e., registers, values), they are expanded by the pre-processor and only viewed as integers by the C compiler. Thus, the mutation rules for identifiers replace an identifier with any other defined identifier. In $C_{Devil}$ code, identifiers also correspond to names of

functions and values of the Devil-generated interface. Mutations for these identifiers are always performed within the same semantic class (*e.g.*, set function, get function).

## 4. Experiments

Our goal is to evaluate the robustness improvement offered by Devil by measuring how early in the development process error are detected. To do so, we have injected programming errors in both Devil specifications and real drivers. By measuring whether or not errors are detected at compile time or at run time, we are able to provide a quantitative evaluation of the Devil approach.

In the rest of this section, we first present an evaluation of the robustness of Devil specifications. Then we compare the robustness of an existing IDE disk driver with the same driver re-engineered using Devil.

### 4.1 Evaluation of the robustness of Devil specifications

In the Devil approach, writing a device specification is a task ideally devoted to a device expert. Still, even experts are prone to typographic and inattention errors. By allowing to verify properties on Devil specification, the Devil compiler checks their consistency and provides support to the expert.

To measure the coverage of the compiler, we have used the mutation rules defined in Section 3.2 to inject errors into five device specifications: the Logitech busmouse, a PCI bus master controller, a IDE disk controller, a NE2000 Ethernet controller, and a graphic card. The results presented in Table 2 show that up to 95% of the mutants injected into a Devil specification are caught by the Devil compiler.

### 4.2 Comparison between the Linux IDE driver and its Devil counterpart

To compare Devil based drivers to pure C drivers, we have chosen to focus on drivers that are critical for the correct behavior of a standard Linux system. We have selected the IDE disk driver, the NE2000 Ethernet controller and the DMA and interrupt controllers. Our current results are only for the IDE driver.

Our study focuses on the code fragments that are activated at boot time. Our mutation model generate about 2000 C mutants, which is more than it is feasible to test exhaustively. Instead, we randomly tested 25% of the generated mutants. Generated mutants are first compiled so as to detect compile-time errors. Then, each mutant that successfully compiles is used to build a test kernel. We have observed the following behaviors when booting such a test kernel.

- 1 - Run-time check. The mutant is detected by a Devil assertion. The kernel prints the source code line number where the error occurs and halts.
- 2 - Dead code. The kernel boots correctly because the mutation occurs in a non-executed path.
- 3 - Boot. The kernel boots. No damage can be observed, suggesting that the mutant has no impact on the kernel up to the end of the boot process.
- 4 - Crash. The kernel crashes but no information is printed. At least a hardware reset is needed to restart the machine.
- 5 - Infinite loop. The kernel loops infinitely and never completes the boot.
- 6 - Halt. The kernel halts and prints a panic message.
- 7 - Damaged boot. The boot completes but there is visible damage (e.g., an unmounted file system, missing files).

Case 1 represents the best situation since the error is detected and an error message clearly identifies the faulty line. Case 2 represents an irrelevant test. Case 3 represents the worst case in a driver development; the kernel contains an error that is likely to show up later. Such an error will be extremely difficult to debug. We distinguish between case 2 and case 3 by careful examination of the execution path. Cases 4 to 7 represent intermediate situations; the driver developer knows the presence of a bug, but tedious tracking is typically needed to isolate the actual source of the error.

| | **Number of mutation sites** | **Number of mutants** | **Concerned mutants / total nb. of mutants** |
|---|---|---|---|
| Compile-time check | 94 | 138 | **26.7** % |
| Crash | 5 | 15 | **2.9** % |
| Infinite Loop | 14 | 58 | **11.2** % |
| Halt | 32 | 111 | **21.5** % |
| Damaged boot | 9 | 15 | **2.9** % |
| Boot | 39 | 179 | **34.7** % |
| **Total** | 94 | 516 | N/A |

**Table 3. Mutations on C code**

Tables 3 and 4 present the results of our mutation analysis on the C and $C_{Devil}$ code of the IDE disk driver. The first line shows how many mutants are detected at compile time. The following lines present the different situations that are encountered while booting a mutated kernel. Finally, the last line presents the number of mutation sites and the total number of mutants in the experiment.

| | Number of lines | Number of mutation sites | Number of injected mutants | % of detected mutants |
|---|---|---|---|---|
| Logitech Busmouse | 22 | 87 | 1678 | **95.4**% |
| PCI Bus Master (Intel 82371FB) | 27 | 82 | 1465 | **88.8**% |
| IDE (Intel PIIX4) | 130 | 352 | 10299 | **91.7**% |
| Ethernet NE2000 (ns8390) | 131 | 434 | 9410 | **92.6**% |
| Graphic card (Permedia 2) | 128 | 400 | 13683 | **90.3**% |

**Table 2. Mutation coverage of the Devil compiler**

| | Number of mutation sites | Number of mutants | Concerned mutants / total nb. of mutants |
|---|---|---|---|
| Compile-time check | 102 | 316 | **58.0** % |
| Run-time check | 23 | 77 | **14.1** % |
| Crash | 0 | 0 | **0.0** % |
| Infinite loop | 3 | 4 | **0.7** % |
| Halt | 13 | 27 | **4.9** % |
| Damaged boot | 3 | 3 | **0.5** % |
| Boot | 31 | 67 | **12.3** % |
| Dead code | 12 | 51 | **9.4** % |
| **Total** | 102 | 545 | N/A |

**Table 4. Mutations on $C_{Devil}$ code**

Our results show that 72% of the errors in the Devil driver are detected either at compile time or at run time. The percentage of mutants detected is even greater if we ignore dead-code mutants. By comparison, only 26.7% errors are detected at compile time on the original C driver. Thus 3 times more errors are detected by Devil mechanisms.

In Devil, only 12.3% of the mutations are not detected (e.g. case 3), while 34.7% of the mutations in the C code are not detected. Thus the worst situation appears 3 times more often in a traditional driver.

It should be noted that dead-code mutants appear only in the Devil based driver. This is due to the $C_{Devil}$ programming style that requires switches for testing the status at the end of each command. In C, the equivalent test is implementing by a contorted single line macro that is always executed.

Finally, each experiment takes about 2 minutes if no manual damage repair has to be done. In fact, two mutants of the original IDE driver crashed the partition table/filesystem and required re-formatting the disk. This effect never appeared for the Devil driver.

## 5. Related Work

Detecting bugs as early as possible is crucial during the development process. A study by DeMillo and Mathur found that typographic and inattention errors represent a significant fraction, though not the majority, of the errors in production programs. This study also revealed that such errors can remain hidden for a long time. Even though their study was concerned with the development of TEX, which differs from device drivers, these observations remain pertinent, and are even more important considering the permissive nature of a language such as C [6].

In previous papers, we described the design of the Devil language [12]. Additionally, we showed that the IDE driver and a X11 driver re-engineered using Devil are almost as efficient as the original ones [11]. Also, we reported preliminary experiments based on mutation analysis for measuring compile-time error coverage of Devil and C drivers [11].

The software implemented fault-injection (SWIFI) technique implements the *bit-flip* fault model, which is meant to simulate the results of physical hardware faults. To some extent, SWIFI tools can be used to emulate software faults [10]. However, it has been found that some software faults can not be emulated by any SWIFI tool.

Integration of microkernels into critical embedded computer systems is promising approach. In this context, evaluation of the microkernel robustness is mandatory. The MAFALDA tool provides objective failure data on a candidate microkernel. Using these data, improved error detection mechanisms have been suggested [8].

Improving the robustness of critical software can also be achieved by using formal methods. Methods such as *B* [1] produce an implementation by successive refinements of a model. However, formal proofs are required at each step and these proofs can not always be automated. Furthermore, the quality of the resulting implementation depends on the quality of the model.

## 6. Conclusion and Future Work

Devil is an IDL for specifying the hardware devices. From a device specification, a compiler generates stubs containing low-level code to operate the device.

In this paper, we have presented an evaluation of the robustness improvement offered by Devil. To do so, we have injected programming errors using mutation analyses into Devil-based Linux drivers and the original C drivers. For that, we have defined mutation rules that model programming errors for both Devil and C.

We assess how early errors can be caught in the development process, by measuring whether errors are detected either at compile time or at run time. The results of our experiments on the IDE Linux disk driver show that nearly 3 times more errors are detected in the Devil driver than in the

original C driver. Additionally, 3 times fewer mutants in the Devil driver than in the C driver result in undetected damage within the kernel. These mutants represent the worst case in driver development since these errors will be extremely difficult to debug. Finally, up to 90% of mutants that were injected within Devil specifications were detected by the Devil compiler.

We are currently evaluating the robustness of Devil over several other Linux drivers such as the an Ethernet driver, and the interrupt and DMA drivers. In future work, we hope to improve the effectiveness of compile-time checks. Specifically, we want to build a preprocessor tool that generate a compile-time comparison function for any Devil type.

## Acknowledgment

We thank Renaud Marlet and Charles Consel, who participated in the design of the Devil language. We thank Julia Lawall from DIKU for helpful comments on earlier versions of this paper. We also thank Anne-Françoise Le Meur, Fabrice Mérillon and Jocelyn Frechot, who helped us for the experiments.

This work has been partly supported by the French Ministry of Economy, Finance and Industry under the ITEA DESS contract 004930214, the French Ministry of Research and Technology under the Phenix contract 99S0362, and the French Ministry of Education and Research.

## Availability

The Devil compiler and experiments mentioned in the paper are available at the following web page `http://compose.labri.fr/prototypes/devil`

## References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] H. Agrawal, R. Demillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Centre, Purdue University, West Lafayette, Indiana, March 1989.

[3] A. T. Agree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, USA, September 1979.

[4] E. N. Dekker and J. M. Newcomer. *Developing Windows NT device drivers: A programmer's handbook.* Addison-Wesley, first edition, March 1999.

[5] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Testing, Analysis, and Verification*, pages 142–151. IEEE Computer Society Press, 1988.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[7] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[8] J. Fabre, L. Arlat, M. Rodriguez, and F. Salles. Dependability of COTS microkernel-based systems. Technical report, LAAS Report 00466, October 2000.

[9] M. R. Girgis and M. R. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Proceedings of the ACM SIGSOFT/IEEE Workshop on Software Testing*, pages 64–73. IEEE, 1986.

[10] H. Madeira, D. Costa, and M. Vieira. On the emulation of software faults by software fault injection. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 417–426, New York, NY, USA, June 2000. IEEE.

[11] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for Hardware Programming. In *4th Symposium on Operating Systems Design and Implementation (ler OSDI 2000)*, pages 17–30, San Diego, California, October 2000.

[12] L. Réveillère, F. Mérillon, C. Consel, R. Marlet, and G. Muller. A DSL approach to improve productivity and safety in device drivers development. In *Proceedings of the* $15^{th}$ *IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 101–109, Grenoble, France, September 2000. IEEE Computer Society Press.

[13] A. Rubini. *Linux Device Drivers*. O'Reilly, first edition, February 1998.

[14] L. Réveillère, F. Mérillon, C. Consel, R. Marlet, and G. Muller. The Devil language. Research Report 1319, IRISA, Rennes, France, May 2000.