Fast and Precise Retrieval of Forward and Back Porting Information for Linux Device Drivers

Julia Lawall, Derek Palinski, Lukas Gnirke, Gilles Muller

Fast and Precise Retrieval of Forward and Back Porting Information for Linux Device Drivers

Julia Lawall, Derek Palinski, Lukas Gnirke, Gilles Muller
Sorbonne Universités/UPMC/Inria/LIP6

Abstract

Porting Linux device drivers to target more recent and older Linux kernel versions to compensate for the ever-changing kernel interface is a continual problem for Linux device driver developers. Acquiring information about interface changes is a necessary, but tedious and error prone, part of this task. In this paper, we propose two tools, *Prequel* and *gcc-reduce*, to help the developer collect the needed information. Prequel provides language support for querying git commit histories, while gcc-reduce translates error messages produced by compiling a driver with a target kernel into appropriate Prequel queries. We have used our approach in porting 33 device driver files over up to 3 years of Linux kernel history, amounting to hundreds of thousands of commits. In these experiments, for 3/4 of the porting issues, our approach highlighted commits that enabled solving the porting task. For many porting issues, our approach retrieves relevant commits in 30 seconds or less.

1 Introduction

The Linux kernel evolves rapidly, with around 70,000 non-merge commits accepted per year since 2013. Commits may fix bugs and add new functionalities, but may also change the interfaces between the kernel core and services that run at the kernel level. For example, between Linux 3.8 (February 2013) and Linux 4.9 (December 2016), 2,439 of the 19,473 functions exported to kernel modules were dropped and 10,056 new exported functions were introduced. This rate of interface changes allows the Linux kernel to rapidly address new needs and resolve performance and security bugs.

While the fast rate of change of Linux kernel interfaces has benefits, it poses challenges for developers of services, such as device drivers, that rely on the kernel interface. Such a developer has to target a particular version of the Linux kernel, but any version chosen will be quickly out of date. Furthermore, potential users of the device may rely on earlier kernel versions, due to *e.g.*, local customizations or stability requirements. The tight dependence of device drivers on fast-changing kernel interfaces means that there is a continual need for forward porting driver code to the interfaces supported by more recent kernel versions, and back porting driver code to the interfaces supported by older kernel versions. This requires a lot of effort for device manufacturers who want to support the needs of a range of clients and for device users who rely on specific kernels.

A major challenge in forward or back porting a device driver is to find out where changes are needed and what changes should be performed. Many drivers interact with the kernel interface in similar ways, and thus change examples are likely to be available in the code history. Still, finding these examples effectively requires knowing what to look for. One approach is to compile the driver with the target kernel and take the resulting error and warning messages as a starting point for identifying porting issues. These messages, however, may be redundant, if one error causes the compiler to misinterpret other code, and may be too concise to sufficiently characterize a porting problem. Even when it is possible to pinpoint the porting issues, an even greater challenge is to find relevant examples among the hundreds of commits per day to the Linux kernel. Git,[1] used for change management in the Linux kernel, supports search for a single regular expression within individual changed lines, via the commands `git log -G` and `git log -S`. But particular terms may appear within changed lines for many reasons, not all of which relate to porting issues, and thus git often returns many irrelevant commits.

The difficulty of obtaining relevant information on how to port a driver thus calls for tool support. In this paper, we propose an approach to ease driver porting based on two tools: *Prequel* and *gcc-reduce*. *Prequel* searches

[1] https://git-scm.com/

This paper appeared in the 2017 USENIX Annual Technical Conference.

in a git commit history for commits matching a query. Queries can include constraints on both changed and unchanged lines, allowing Prequel to obtain more precise results than git. Prequel furthermore ranks the resulting commits according to the degree of success of the match, rather than chronologically as done by git, giving the driver maintainer easy access to the most relevant results. *Gcc-reduce* complements Prequel by creating a bridge from the compiler. Given a set of compiler errors and warnings, gcc-reduce reduces them to those that are relevant to porting, and collects complementary information from the source code. Gcc-reduce then generates Prequel queries based on the collected information. Although the possibility remains to write Prequel queries by hand, gcc-reduce is able to generate most queries relevant to driver porting automatically. Overall, our approach permits the developer to save time and effort, by obtaining change examples that are relevant to the porting problem.

The contributions of this paper are as follows:

- Via a case study, we identify two key challenges in driver porting: determining 1) where changes are needed and 2) what changes should be performed.
- We propose the tools Prequel and gcc-reduce that automatically collect information to address these challenges.
- We evaluate Prequel and gcc-reduce by porting 33 device driver files introduced into the Linux kernel in 2013 or 2015 to or from Linux 4.6, released in May 2016. Our approach provides information from the git commit history that enables us to carry out the port for 3/4 of the issues encountered.
- We show that our approach is suitable for use on a standard laptop, with many patch queries completing in 30 seconds or less.
- We compare our approach to the use of git to query the commit history and the use of Google to find relevant change suggestions. For queries such as field type changes, we find that git returns many irrelevant commits. For only 33% of the issues does Google return possibly relevant results among the top 6 entries in the query summary page. In contrast, for our ported 33 driver files, the top ranked commit returned by our approach is helpful for 86% of the porting issues.

2 Motivating Example

To understand the challenges in obtaining adequate information on how to carry out a driver port, we consider the lms501kf03 TFT LCD panel driver, introduced into the Linux kernel in February 2013 in the commit 1be9ca2, and first released in Linux 3.9. The driver consists of a single `.c` file, drivers/video/backlight/lms501kf03.c. We forward port this code over 16 Linux kernel releases, to Linux 4.6, released in May 2016.

The experiment. Compiling the original driver with the Linux 4.6 kernel[2] produces the errors and warnings shown in Figure 1. There are two errors (lines 1 and 8), about the `suspend` and `resume` fields being unknown, and six warnings. The warnings appear to be triggered by the same cause as the errors, and thus we focus on the latter. Specifically, we need to find examples of how to remove `suspend` and `resume` fields, and then see what we can infer from those examples for porting our driver.

Focusing on `suspend`, we can try the following git command, considering commits between the kernel version originally targeted by the driver and the version that is the target of the port:

```
git log -p -G "\<suspend\>" --diff-filter=M 1be9ca2..v4.6
```

`-p` prints the changed lines, `-G "\<suspend\>"` restricts the results to commits that contain the word `suspend` on a changed line, and `--diff-filter=M` restricts the results to commits that perform modifications, as opposed to adding or removing files.

Despite the relative sophistication of this git command, many of the results are completely irrelevant. For example, the first result, commit ba41e1b, removes a reference to a `suspend` field and adds another such reference on the same structure. Such a commit does not help fix a reference to a field that no longer exists. Rather, we need commits in which `suspend` appears on removed lines, but not added ones, which is not expressible with `git log -G`. Subsequent commits give similar results. Indeed, these commits are modifying `suspend` fields in structures having types different from the one, `spi_driver`, used in our driver.

Ideally, we could extend the `git log -G` to include the name of the structure type, but this type name is not likely to be on the same line as the field reference. Nevertheless, we can use the search command of the git viewer to find occurrences of `spi_driver` in the context lines of the emitted patch code. This process might not succeed, because the type name can be arbitrarily distant from the changed lines. In our case, though, it is successful, but the user has to analyze and scroll over 7 occurrences of `spi_driver` within 569 commits before reaching a relevant commit 9d9f780, from January 2015.

Figure 2 shows extracts of the commit 9d9f780. This commit removes initializations of the `suspend` and `resume` fields, on lines 37-38, but also does many other

[2] `make drivers/video/backlight/lms501kf03.o`, with gcc (Ubuntu 4.8.4-2ubuntu1 14.04.3) 4.8.4.

```
drivers/video/backlight/lms501kf03.c:433:2: error: unknown field ‘suspend’ specified in initializer
  .suspend = lms501kf03_suspend,
  ^
drivers/video/backlight/lms501kf03.c:433:2: warning: missing braces around initializer [-Wmissing-braces]
drivers/video/backlight/lms501kf03.c:433:2: warning: (near initialization for ‘lms501kf03_driver.driver’) [-Wmissing-braces]
drivers/video/backlight/lms501kf03.c:433:2: warning: initialization from incompatible pointer type [enabled by default]
drivers/video/backlight/lms501kf03.c:433:2: warning: (near initialization for ‘lms501kf03_driver.driver.name’) [enabled by
      default]
drivers/video/backlight/lms501kf03.c:434:2: error: unknown field ‘resume’ specified in initializer
  .resume  = lms501kf03_resume,
  ^
drivers/video/backlight/lms501kf03.c:434:2: warning: excess elements in struct initializer [enabled by default]
drivers/video/backlight/lms501kf03.c:434:2: warning: (near initialization for ‘lms501kf03_driver’) [enabled by default]
```

Figure 1: Messages resulting from compiling the original Linux lms501kf03 TFT LCD panel driver with Linux 4.6

things. We thus next have to determine whether these other changes are relevant to our driver and whether other examples are needed. We focus on the changes in the definitions of the functions `as3935_suspend` and `as3935_resume` that are stored in the `suspend` and `resume` fields, respectively. The parameter lists of these functions are modified to change the type of the first parameter, and, in the case of the suspend function, to drop the second parameter. In the body of each function, a call to `spi_get_drvdata` on the original first parameter is replaced by a call to `dev_get_drvdata` on the new first parameter (lines 10-11 and 16-17). No change is made to compensate for dropping the second parameter of the suspend function, as it is unused.

We next compare the observed set of changes to the code found in the TFT LCD panel driver that we want to port. Figure 3 shows the relevant code fragments. Some code fragments are analogous to the ones modified in commit 9d9f780. For example, our driver also initializes `spi_driver` `suspend` and `resume` fields to locally defined functions, `lms501kf03_suspend` and `lms501kf03_resume`, respectively. These functions have the same list of parameters as found in commit 9d9f780, and the function stored in the `suspend` field again has the property that the second parameter is not used. The associated changes found in the sample commit can thus be applied directly.

The set of changes illustrated in commit 9d9f780 are, however, insufficient for determining how to update the function bodies. While the suspend and resume functions affected by commit 9d9f780 use their first parameter only in calls to `spi_get_drvdata`, the suspend and resume functions in our driver call `dev_get_drvdata` and `dev_dbg` on values derived from this parameter. Thus, we need more examples. Searching further through the commits, we find commit 01f9326 from March 2013 that is similar to the commit shown in Figure 2 but contains the following change in the functions stored in the `suspend` and `resume` fields:

```
-  struct snd_card *card = dev_get_drvdata(&spi->dev);
+  struct snd_card *card = dev_get_drvdata(dev);
```

```
diff --git a/drivers/iio/proximity/as3935.c ...
index 466aa43..bc0d68e 100644
--- a/drivers/iio/proximity/as3935.c
+++ b/drivers/iio/proximity/as3935.c
@@ -273,9 +273,9 @@ static void calibrate_as3935(...)
 #ifdef CONFIG_PM_SLEEP
-static int as3935_suspend(struct spi_device *spi,
-      pm_message_t msg)
+static int as3935_suspend(struct device *dev)
 {
-  struct iio_dev *indio_dev = spi_get_drvdata(spi);
+  struct iio_dev *indio_dev = dev_get_drvdata(dev);
@@ -293,9 +293,9 @@ err_suspend:
-static int as3935_resume(struct spi_device *spi)
+static int as3935_resume(struct device *dev)
 {
-  struct iio_dev *indio_dev = spi_get_drvdata(spi);
+  struct iio_dev *indio_dev = dev_get_drvdata(dev);
@@ -311,9 +311,12 @@ err_resume:
+
+static SIMPLE_DEV_PM_OPS(as3935_pm_ops, as3935_suspend,
+      as3935_resume);
+#define AS3935_PM_OPS (&as3935_pm_ops)
+
 #else
-#define as3935_suspend NULL
-#define as3935_resume  NULL
+#define AS3935_PM_OPS NULL
 #endif
@@ -441,12 +444,11 @@ static struct spi_driver as3935_driver
   .driver = {
     .name   = "as3935",
     .owner  = THIS_MODULE,
+    .pm     = AS3935_PM_OPS,
   },
   .probe    = as3935_probe,
   .remove   = as3935_remove,
   .id_table = as3935_id,
-  .suspend  = as3935_suspend,
-  .resume   = as3935_resume,
 };
```

Figure 2: Example update on suspend and resume. Some context lines are omitted for readability.

Commit eba3bfb from April 2013 illustrates the case of a `dev_dbg` call:

```
-  dev_dbg(&spi->dev, "lcd->power = %d\n", lcd->power);
+  dev_dbg(dev, "lcd->power = %d\n", lcd->power);
```

All these changes, from these different commits, provide a model for the porting of the TFT LCD panel driver.

```
#if defined(CONFIG_PM)
static int lms501kf03_suspend(struct spi_device *spi,
      pm_message_t mesg)
{
  struct lms501kf03 *lcd = dev_get_drvdata(&spi->dev);
  dev_dbg(&spi->dev, "lcd->power = %d\n", lcd->power);
  return lms501kf03_power(lcd, FB_BLANK_POWERDOWN);
}
static int lms501kf03_resume(struct spi_device *spi)
{
  struct lms501kf03 *lcd = dev_get_drvdata(&spi->dev);
  lcd->power = FB_BLANK_POWERDOWN;
  return lms501kf03_power(lcd, FB_BLANK_UNBLANK);
}
#else
#define lms501kf03_suspend NULL
#define lms501kf03_resume NULL
#endif

static struct spi_driver lms501kf03_driver = {
  .driver = {
    .name = "lms501kf03",
    .owner = THIS_MODULE,
  },
  .probe = lms501kf03_probe,
  .remove = lms501kf03_remove,
  .shutdown = lms501kf03_shutdown,
  .suspend = lms501kf03_suspend,
  .resume = lms501kf03_resume,
};
```

Figure 3: lms501kf03 TFT LCD panel driver extract

Assessment. Our example illustrates clearly that compiler error messages are helpful, but there can be a significant difference between the set of the errors raised by the compiler and the set of changes required. In Figure 1, we see that the compiler may report errors and warnings that are actually side-effects of other issues, and do not help to identify the set of changes required. Furthermore, many changes, such as the changes in the definitions of `lms501kf03_suspend` and `lms501kf03_resume`, are required but were not reported by the compiler, and thus adequate examples of changes on similar drivers are necessary to determine what changes are needed and how to carry them out. Then, finding even one commit that illustrates a specific problem is a major challenge. As no one commit may illustrate all of the issues relevant to a particular driver, repeating this process to find multiple commits may be necessary.

3 Prequel

The core of our approach is the process of searching for commits that illustrate how to perform a particular kind of change. Such a search must be able to take into account properties of both changed lines and the context in which the changed lines occur. For example, we would like to retrieve only commits that remove initializations of `suspend` and `resume` fields that are in `spi_driver` structures. To retrieve such commits, we propose a patch query language, PQL, that permits describing properties

```
1 @bad depends on after@
2 identifier i;
3 expression e;
4 @@
5 struct spi_driver i = {
6   .suspend
7     = e,
8 };
9 @rem depends on !bad@
10 identifier i;
11 expression e;
12 @@
13 struct spi_driver i = {
14 - .suspend
15     = e,
16 };
```

Figure 4: Patch query detecting removals of the `suspend` field from the initialization of a `spi_driver` structure

of both changed lines and their context and a tool Prequel that applies PQL queries to a git commit history. We first briefly present the syntax and semantics of PQL. More details are available in a technical report [5]. We then describe optimizations that allow using Prequel on a standard laptop.

3.1 PQL syntax and semantics

To describe changes and their context in systems code, we take inspiration from the program transformation tool Coccinelle [2, 8]. Coccinelle and its Semantic Patch Language (SmPL) offer a transformation language based on the familiar patch syntax. Coccinelle is widely used in Linux kernel development, and its notation is familiar to kernel developers. Our key insight is that a specification of which lines to add and remove can also be viewed as a description of the lines that have been added and removed, after the transformation has been performed. We thus propose a SmPL-like notation for PQL, providing a description, which we refer to as a *patch query*, of the effect of a previous transformation process. Prequel then applies a patch query to a series of commits.

Figure 4 shows a patch query that detects commits that remove an initialization of a `suspend` field in a `spi_driver` structure. The patch query consists of two rules, the rule `bad` on lines 1-8 followed by the rule `rem` on lines 9-16. We first focus on the latter. A patch query rule consists of a fragment of code that combines concrete terms, such as the type name `spi_driver`, with *metavariables*, declared at the top of the rule. `rem` uses metavariables for the name of the driver structure, `i` (line 13), which is declared to match any identifier (line 10), and the initial value of the `suspend` field `e` (line 15), which is declared to match any expression (line 11). Within the code fragment, `-` and `+` indicate tokens that must be removed or added by a matching commit, respectively. `rem` indicates that the token `suspend` must be removed (line 14).

The output of gcc indicates that we want to find commits that remove the `suspend` field, but does not indicate what other changes may be needed. For example, one way to remove a field is to rename it, in which case the initial value expression `e` may remain unchanged, while

another possibility is to remove both the field name and the initial value expression entirely. Prequel is designed based on the hypothesis that a user searches for a change to complete his understanding of that change, and thus Prequel provides approximate matching. Specifically, tokens that are annotated with - or + *must* be removed or added, respectively, but other tokens *may* be removed or added as well. To distinguish between more or less precise matches, Prequel returns the matching commits, ranked by the percentage of changed lines or hunks that contain an exact match of the specification.

The rule `rem` also does not guard against the possibility that the `suspend` field is simply moved around in the `spi_driver` structure, *i.e.*, removed but added back, or that one instance of a `suspend` field is removed but others remain. As gcc reports that `suspend` is unknown, we need to find example commits that remove the `suspend` field from the `spi_driver` completely. The rule `bad` (lines 1-8) extends the patch query to ensure this property. This rule matches a commit for which the state of the code after the commit, as indicated by `depends on after` on line 1, contains an initialization of a `suspend` field. Such commits are ones that we do *not* want to see. The rule `rem` then depends on the failure of the rule `bad`. Prequel returns only commits that match rules on which no other rules depend, and thus the only results are those that completely remove the `suspend` field (rule `rem`).

3.2 Optimizations

We want to support driver porting on a standard laptop, as would be most easily accessible to a kernel developer. The matching performed by Prequel, however, may be very expensive. Because a patch query may describe not only changed code, but also context code that occurs elsewhere in the same file, *e.g.*, the type `spi_driver` in our case, Prequel matches a patch query against complete files as they exist before and after a commit, and not just against the changed lines. Matching a query against all the files affected by hundreds of thousands of commits, as are found in several years of history of the Linux kernel, however, would be very time consuming. In practice, due to the diversity of the Linux kernel, for any given set of keywords, only a small percentage of the commits are likely to be relevant. Thus, performing such thorough matching is often unnecessary.

To reduce the set of commits considered in detail, Prequel first analyzes a patch query to identify keywords that must be present in or near the changes made by any commit that the patch query can match. For example, in the patch query of Figure 4, Prequel would select `suspend` as a keyword that must be present in the lines removed by a commit to allow a match. Prequel then searches for these keywords in the patch associated with each commit and ignores commits where they are not found, exploiting the fact that a patch is typically much smaller than the affected source files. Alternatively, for better performance, the user can prepare indices in advance, using the GNU utility ID Utils.[3] These indices map tokens to 1) the commits for which they occur on removed lines 2) the commits for which they occur on added lines, and 3) the commits for which they occur on changed lines or within 3 lines of context code (the default when using the `diff` command). Prequel uses the first two indices to identify commits that contain tokens annotated in the patch query with - or +, respectively, while it uses the third index to identify commits that contain unannotated tokens. Using an index is a choice left up to the user, because it trades flexibility in the range of considered commits for performance. Currently, the Prequel distribution includes indices for Linux versions 3.0 through 4.6, the range considered in our evaluation.

Limiting the set of commits considered based on keywords found in patches is only effective when the relevant keywords are found within or near the changed lines. Some relevant keywords, however, may appear far from any change. For example, in our case, if the `suspend` field was not dropped, but rather had its type changed, then the patch could be on the definition of the function stored in the `suspend` field, which could be far from any mention of `suspend`. When important keywords are expected to be far from changed lines, Prequel collects all of the files that contain the keywords in a reference version of the Linux kernel chosen by the user, *e.g.*, the source or target version of the port, and then considers only the commits that affect these files. It is also possible to provide an index mapping tokens to the reference version files in which they occur, to further improve performance.

The above strategies may incur false negatives: a keyword may appear as required in the code before or after the commit, but not close enough to changed lines, or a keyword may not appear anywhere in the reference version. To select commits, Prequel first tries keywords annotated with - or +, which must be within the changed lines, and only relies on unannotated keywords if consideration of the annotated keywords does not sufficiently reduce the number of commits to analyze in detail.

4 Gcc-reduce

Extracting the relevant information from compiler errors and from the source code to create Prequel patch queries is tedious and error prone. Our tool gcc-reduce provides a front end to Prequel for driver porting that automates this task. Because the kinds of error messages that the

[3] https://www.gnu.org/software/idutils/

compiler generates are limited, gcc-reduce can also automate the construction of patch queries, in most cases.

Gcc-reduce collects information required for the search for change examples from the errors and warnings[4] produced by compiling the driver with the target version and from the driver source code. Gcc-reduce expects the use of gcc for compilation; LLVM is known to give better error messages, but the support for compiling the kernel with LLVM is incomplete, and appears to be not well maintained.[5] Gcc-reduce then 1) reduces the resulting compiler errors to those that indicate porting issues, and 2) generates PQL patch queries from the information collected from the compiler error messages, as described below.

4.1 Error message reduction

As illustrated in Figure 1, gcc often gives multiple error messages that actually derive from the same problem. Three issues arise: 1) a problem recurs, 2) dataflow relationships imply that a problem in one part of the code makes another part of the code invalid, 3) the same as the second case, but triggered by structural relationships rather than dataflow.

To characterize the compiler errors, we have created a number of *error categories*, such as "unknown field error", illustrated by lines 1-3 of Figure 1. First, to detect recurring errors, for each error, gcc-reduce selects the corresponding error category and collects keywords that uniquely identify the problem. The keywords can come from the error message itself or from the source code. For example, for the error on lines 1-3 of Figure 1, the keywords are `suspend`, obtained from the error message and representing the affected field, and the name of the type of the enclosing structure, `spi_driver`, which is obtained from the source code. `gcc-reduce` discards subsequent errors of the same error category with the same keywords.

Second, analogous to the notion of *gen* and *kill* sets in dataflow analysis [1], gcc-reduce collects for each error an *input* set, containing keywords that, if they have been associated with a previous error, imply that the current error is redundant, and an *impact* set, containing keywords that, if they are associated with a future error, imply that the future error is redundant. In our example, the input set is {`struct spi_driver.suspend`, `struct spi_driver`}; if some other error has been reported related to the `suspend` field of a `spi_driver` structure, then resolving that error is likely to also resolve the one in our example, and the error in our example is not needed. Furthermore, if the entire `spi_driver` structure has been found to be invalid, then there is no need for an error about one of its fields. The impact set then is {`struct spi_driver.suspend`}, indicating that a problem has been found with the `suspend` field of a `spi_driver` structure, and thus no other messages, of any kind, that derive from use of this field, *i.e.*, that contain this field in their input set, are needed.

Finally, various kinds of problems can trigger errors that relate to the code structure. An example is the error about missing braces in line 4 of Figure 1. Our hypothesis is that the driver to port compiles correctly with its original Linux version, and thus such structural errors should be side effects of other errors. For certain kinds of messages, all other errors of certain kinds found in the same block or function are discarded.

In our example, gcc-reduce retains only the error on lines 1-3 and the one on lines 8-10 for further processing.

4.2 Patch query generation

After reducing the error messages, gcc-reduce creates a patch query for each of the remaining errors. The various kinds of errors are limited, as are the kinds of information found in their keywords. Accordingly, the patch queries can be generated by instantiating a small set of templates. 17 templates, incorporating PQL best practices, are available in our current prototype. Templates are typically parameterized by type names, *e.g.*, `spi_driver` in our example, and global function and field names, *i.e.*, terms that are common to the kernel rather than specific to the driver. gcc-reduce also generates a makefile and a document that the maintainer can use to track the changes required. A few error types are not supported by our current set of templates. In these cases, the driver maintainer can study the provided templates and produce a patch query by analogy.

5 Evaluation

The goals for our evaluation are to assess the degree to which our approach satisfies the following properties:

- Our approach is efficient enough for interactive use on a standard laptop.
- `gcc-reduce` eliminates redundant compiler error messages, but keeps the errors needed to motivate a complete forward or back port of a driver.
- The commits selected by Prequel help solve forward and back porting problems.

[4] Subsequently, we refer to compiler errors and warnings collectively as errors.

[5] http://llvm.linuxfoundation.org/index.php/Main_Page; The section "Current patch statistics" is dated 2015-01-28.

- Our approach gives more relevant results than existing approaches, such as commit history search using git or Internet search using Google.

Our evaluation focuses on drivers introduced in 2013 and 2015 and targets Linux 4.6, released on May 15, 2016, to illustrate the behavior of our approach over a longer and shorter time period. From January 1, 2013 to May 15, 2016 there were 219,879 commits to the Linux kernel and from January 1, 2015 to May 15, 2016 there were 85,812 commits to the Linux kernel. We first present our dataset and then address the above properties.

5.1 Dataset

We consider an introduced driver to be a collection of C and header files that are added into the kernel in the `drivers` directory in a single commit, accompanied by changes in a Makefile and other files related to the build infrastructure. We ignore `drivers/staging` code, as such drivers are considered to be immature and thus may contain idiosyncratic coding strategies for which example changes may not be available. We check also that the driver compiles without errors or warnings at the point where it is committed and that all of the files added or modified by the commit exist in our target version, Linux 4.6. Finally, for our forward porting experiments, for each selected commit, we overwrite the corresponding C and header files in a clean Linux 4.6 and force the compilation of each C file. We include in our data set drivers for which this compilation produces at least one warning or error. Likewise, for our back porting experiments, we take the Linux 4.6 versions of the driver files introduced in 2013 or 2015 and copy them back to the kernel version just following the commit in which the files were introduced, keeping for further analysis the files in which compilation with the older version produces errors.

Table 1 shows the number of drivers and driver `.c` files that raise porting issues. Table 2 shows the distribution of these drivers over the various driver types. The difference in the number of drivers and driver files considered in the forward and back porting cases is due to the latency of interface deprecation in the Linux kernel. An outdated and a modern interface may coexist in a newer or older kernel, in which case forward porting or back porting, respectively, is not necessary. Our approach only aims to produce a driver that is compatible with the target kernel version, and does not aim to ensure that the resulting driver uses the most recent interfaces, if the interfaces from the source kernel version remain available.

We use the complete set of drivers in our dataset for the evaluations that are fully automatic. For our porting experiments, we use only a subset, due to time constraints.

Table 1: Drivers and driver files that raise porting issues

	forward port		back port	
	drivers	driver files	drivers	driver files
2013	108	135	130	149
2015	97	119	114	125

Table 2: Distribution of considered driver files (f = forward ports, b = back ports)

	2013f	2013b	2015f	2015b		2013f	2013b	2015f	2015b		2013f	2013b	2015f	2015b		2013f	2013b	2015f	2015b
acpi	0	0	1	0	fmc	1	1	0	0	mailbox	0	0	0	2	powercap	1	0	0	0
ata	1	1	0	1	gpio	5	5	5	7	md	3	2	2	2	pwm	1	1	0	0
base	0	0	0	1	gpu	6	1	6	8	media	6	6	5	2	regulator	5	4	0	0
block	5	3	0	0	hid	0	0	1	1	mfd	2	4	0	4	reset	0	0	0	4
char	3	4	0	1	hwmon	0	3	1	1	misc	1	1	0	0	rtc	2	10	0	0
clk	6	6	10	15	hwtracing	0	0	2	3	mmc	1	2	2	2	scsi	3	2	3	1
clocksrc.	3	4	2	3	i2c	2	4	0	1	mtd	0	0	5	3	soc	0	0	2	1
cpufreq	1	1	0	0	iio	22	32	2	2	net	14	8	18	14	spi	3	1	0	0
crypto	0	0	1	2	infiniband	6	0	0	0	nfc	2	3	4	5	thermal	5	6	6	6
devfreq	0	0	1	1	input	3	5	1	1	phy	4	2	2	2	usb	1	2	1	1
dma	0	1	0	3	irqchip	0	0	1	1	pinctrl	2	1	3	6	video	1	4	0	0
edac	0	1	0	0	leds	0	1	3	2	platform	4	3	2	2	virtio	0	0	0	1
extcon	0	0	3	2	lightnvm	0	0	3	3	power	6	8	8	4	watchdog	0	2	2	5

5.2 Methodology

For a chosen driver, we apply our approach to collect relevant commits. From these commits, we manually infer the required changes and update the original driver accordingly. We consider an experiment to be a success if the changes we have made in the driver are also found in the target version of the driver and the resulting driver compiles in the target version. We do not aspire to produce code identical to the target version, because the driver may also undergo changes that are specific to its behavior, which go beyond the porting task.

We prepare indices for Prequel covering all commits that modify at least one file between Linux 3.0, released in July 2011, and Linux 4.6, amounting to 306,242 commits. The largest index is the third (Section 3.2), at 160MB. Using indices starting with 2011 is overkill when porting drivers from 2013 at the earliest. We envision, however, that developers will generate new indices only occasionally, and thus may have available a larger index than the minimal one needed for a given task. We also prepare an index of our reference version (see Section 3.2), Linux 4.6. This index is used when a keyword is needed to reduce the set of commits to which a Prequel query should be applied, but there is no keyword in a patch query that is expected to occur in or near a changed line.

5.3 Performance

Figure 5 shows the execution times of the most time-consuming parts of Prequel: commit selection and application of the patch query to the selected commits. Experiments are carried out on a single core of an Intel i5-

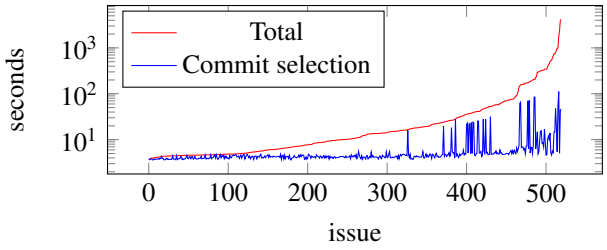

Figure 5: Execution time of Prequel on 2013 forward port issues (Total = patch query application time + commit selection time)

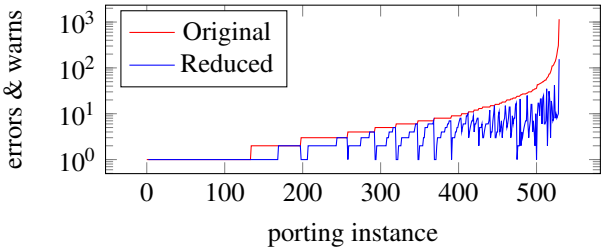

Figure 6: Reduction in the number of errors and warnings achieved by gcc-reduce

6200U 2.30GHz CPU. The machine has 12G memory. The Linux kernel and all temporary files are placed in an in-memory file system to avoid disk access costs. Each point in the graph represents a single porting issue. Issues are ordered from shortest execution time to largest. At each x-axis coordinate, the height of the blue (lower) point represents the commit selection time, and the difference between the red (upper) point and the blue point represents the patch query application time.

For 73% of the issues, the total of commit selection time and patch query application time is 30 seconds or less. Beyond that point, the commit selection time is occasionally high, up to 111 seconds, typically when the reference version is used as a last resort to reduce the number of possible commits. The average commit selection time is under 7 seconds. Patch query application time rises with the number of files in the selected commits and the file size. The maximum patch query time per commit is under 14 seconds. Further performance improvements require finding low cost ways of discarding more commits, before applying the patch query.

The overall running time of the approach for a particular driver depends on the number of issues involved. This is managed by gcc-reduce. Figure 6 shows in the red (top) line the number of errors and warnings indicated by gcc for all of the considered porting problems (2013 and 2015 drivers, forward and backward porting) and in the blue (bottom) line the number resulting from the reduction process. Each point represents a driver, and the drivers are ordered from the smallest to the largest number of gcc messages. 36% of the ports involve only one issue. 76% involve fewer than 5. For a driver with 4 issues, each taking 30 seconds or less, we thus obtain a typical patch query time of 2 minutes or less.

Finally, Table 3 shows the distribution of error types, for driver subdirectories with more than 30 errors in at least one porting experiment, as well as the average commit selection and patch query application time (Avg PQ) for each error type, as observed on the 2013 forward porting experiments (Figure 5).

Table 3: Distribution of error types

	Unknown	Unknown function	Unknown variable	Unknown type	Unknown field	Arg error	Expression type change	Field type change
char	1	22	43	4	8	4	0	0
clk	104	47	65	2	12	3	0	5
gpio	1	36	15	1	30	3	4	0
gpu	1	18	7	20	16	19	0	8
iio	13	64	31	24	35	21	3	25
infiniband	1	2	4	11	17	2	0	3
lightnvm	2	7	7	1	26	10	0	2
misc	10	16	6	1	2	4	0	0
net	6	55	49	20	55	15	1	21
platform	2	20	17	31	11	4	0	0
power	6	12	3	16	60	24	0	0
Avg PQ (sec)	N/A	25	7	9	57	197	14	267

5.4 Precision

We have used our approach in the porting of 33 driver files: 13 from their original versions in 2013 to Linux 4.6, 10 from Linux 4.6 back to their commit of introduction in 2013, and 10 from their original versions in 2015 to Linux 4.6. Gcc-reduce reduced the compiler errors and warnings associated with these drivers to 107.

For 80 of the identified issues, we were able to replicate the change as found in the Linux kernel code. For 24 issues, we encountered some kind of failure. In 6 cases, gcc-reduce misclassified an issue. For example, for two issues related to forward porting the 2013 Xen TPM frontend driver, introduced in commit e268395, gcc-reduce expects a change in the field in the case of an incompatible field initialization, while the actual issue is a change in declaration of the initial value. Gcc-reduce could create patch queries that consider more possibilities at the cost of a higher patch query time. In the same driver, there is also the only instance of a failure of Prequel: the issue is related to a macro whose uses Prequel is unable to parse, implying that no results are returned.

In 5 cases, there is more than one change to a particular code fragment between the original version of the code and the target of the porting task. For example, the changes to the Sharp GP2AP020A00F Proximity/ALS sensor driver between its original version in 2013 (commit bf29fbe) and Linux 4.6, in part involve first append-

ing _new to a set of structure fields, to allow old and new versions of the fields to co-exist, and then removing the _new in a later commit, once all relevant drivers have been updated. Prequel only finds the commit that adds the _new suffix, thus giving only a partial view of the required changes. In the remaining failure cases, the commits available are not sufficient to decide how to transform a particular piece of code. For example, back porting the Nuvoton NAU7802 ADC driver from its Linux 4.6 version to the Linux kernel as of commit 8b20be8 involves removing calls to `reinit_completion`. Many examples are available, but they involve different transformations, and indeed the commit log message indicates that many of them are bug fixes. Using a bug fix as a model for backporting would amount to introducing a bug. Thus, it is not clear from the examples which strategy is appropriate for the given driver.

Overall, we were able to address 3/4 of the issues successfully, without having any specific prior knowledge of the drivers concerned. Furthermore, doing so typically required looking at very few commits. For 86% of the successfully addressed issues, it was sufficient to look at only the first reported commit. We consulted at most 7 commits for a single issue.

We have also found back porting to be harder than forward porting. We have already noted the case where many examples are bug fixes. Furthermore, over time, the Linux kernel developers also tend to replace local, special-purpose coding strategies by generic APIs, when some operations or data are common to multiple drivers. We have often found it easier to introduce generic code from specific implementations, as required for forward porting, but harder to replace API calls by specialized local definitions or data structure representations where the design strategy may be specific to each developer.

5.5 Comparison with git

Git is at the foundation of many Linux kernel developers' development practices. We thus compare the information obtained by Prequel with the information that can be obtained using `git log -G` or `git log -S`.[6] We consider only the driver files for which we have carried out the full forward porting process (Section 5.4) and only the issues that we resolved successfully. In each case, we collect the commits that reference a keyword that is expected to be changed by the port; for example if the issue is an unknown `suspend` field, then we expect `suspend` to appear in the changed lines. We consider a range starting with the commit that we found most helpful and ending with Linux 4.6, to see how many commits git provides to the user before reaching the helpful one.

[6] `git log -G` finds patterns in changed lines, while `git log -S` additionally requires that the number of instances of those patterns changes. We choose an appropriate command given the error type.

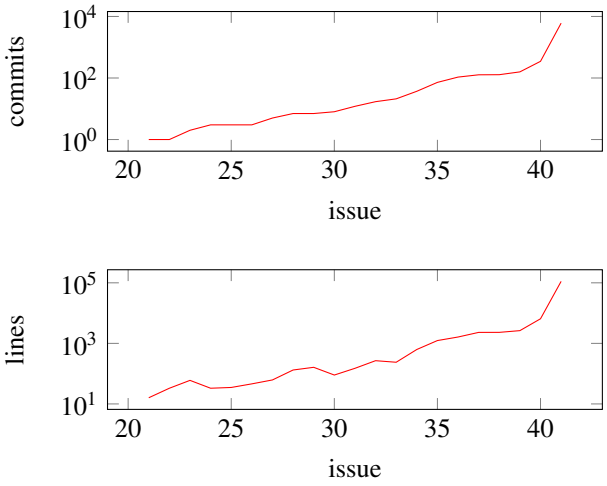

Figure 7: Commits (top) and commit lines (bottom) obtained with git log -G or git log -S

Each git command took around 90 seconds. Figure 7 shows the number of commits returned (top) and the number of lines in these commits (bottom), including both log messages and code changes. Issues are ordered by increasing number of commits, in both cases. For half of the issues (1-20 on the x-axis) there are no results, and thus these points are omitted. No results is typical of issues such as an unknown function; once the function has been removed, no more commits will mention it. Git thus succeeds in immediately returning a commit relevant to the porting problem. Other kinds of issues such as a change in the number of arguments of a function or the type of a structure field, however, do not cause the keyword to disappear, and thus it can occur in later, irrelevant commits. Furthermore, as noted in Section 2, some names, typically those of structure fields, are reused across the kernel, and thus commits are found affecting unrelated instances. For example, the rightmost point in each graph of Figure 7 derives from a search for `dev`, which is very common both as a field name for many types of structures and as the name of a local variable. Prequel on the other hand has access to type information and other relevant context information, and can select commits more precisely. In many cases, Prequel is even more efficient than git, due to the use of indexing.

Finally, 8 issues concern the incorrect type of the initial value of a structure field. In these cases, the change is typically in the definition if the initial value, which is often driver-specific, and there is no keyword whose changes git can be used to search for.

5.6 Comparison with search engines

Our approach to driver porting relies on searching through the commit history for information on how to resolve compiler errors resulting from out of date code. In many other software development contexts, developers and users turn to search engines such as Google for hints on how to address error messages. To assess the potential benefits of using an existing general-purpose search engine to address Linux driver porting issues, we have done a small experiment using Google.

Setup. It is impossible to anticipate every Google query that a developer might make. We take the straightforward solution of using the compiler error message itself (see Figure 1) as a query. In this, we drop the position information (file, etc.), which is likely too restrictive, unless someone has already ported the same driver.

Some error descriptions, however, are generic, such as the message "initialization from incompatible pointer type", found in the middle of Figure 1, that does not contain any information specific to the error context. As illustrated in Figure 1, gcc error messages either contain a code snippet, or implicitly inherit the code snippet of a previous message. Thus, we additionally consider concatenating the error description and the code snippet to form another possible query, providing more information but incurring the risk of overspecification to the targeted driver. We consider only the subset of error messages generated by our error message reduction process (Section 4.1), as the user could manually filter out the important messages, as done by our tool.

In order to assess the information provided by Google on a large scale, we use the `curl` library [3] to script requests. We then parse each resulting search result summary page to extract the entries, consisting of a title, the link as shown to the user, and the description, found on the first page of results. These tests involve our complete dataset (Section 5.1).

To assess the results, we use the measure *query recall rate at N*, meaning the percentage of cases for which the top N Google results contain at least one result that is relevant to the porting problem. A result is considered relevant if it contains all of the keywords identified by gcc-reduce for the issue. As shown in Figure 8, based on this criterion, the first result appears to be relevant only 12% of the time in the no code case, where we include only the error message. The rate of at least one relevant result rises to only 33% in the no code case if one considers the top 6 results. Even a 33% success rate is not very useful in practice, and even if the information on the summary page suggests success, there is no guarantee that the information on the linked page will actually turn out to be useful. In contrast, for the ports we have carried

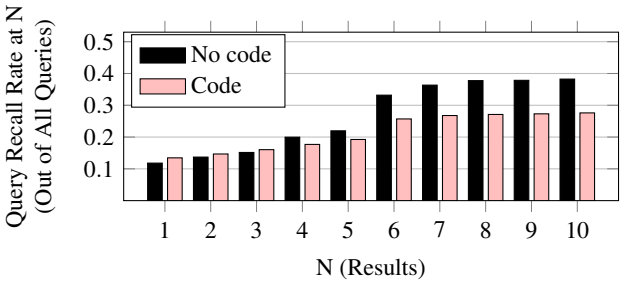

Figure 8: Rate of existence of a relevant result in the top N returns returned by Google

out, in 86% of the cases, we resolved an issue by looking only at Prequel's top ranked commit. Furthermore, there is contradictory feedback about whether it is desirable to include the code snippet; when looking at the top 1-3 results, doing so slightly increases the rate of success, but when considering a larger set of results, including the code snippet is a disadvantage. So the driver maintainer may have to launch several queries to see which is the best one. Overall, general-purpose search engines do not appear to be a promising tool for finding information about Linux kernel interface changes.

6 Limitations

Our approach is based on several hypotheses. First, we assume that the driver to port works correctly with respect to the Linux kernel version for which it has been developed. Thus, the goal of porting is to preserve its behavior. Second, we make a similar assumption about the commits we encounter; their changes preserve correctness. Third, only one change is required per issue to reach code compatible with the target version. Fourth, all relevant issues are highlighted by gcc.

We have encountered violations of the last three hypotheses in practice. A violation of the second hypothesis is illustrated by the `reinit_completion` case in Section 5.4; commits may fix bugs rather than preserve correctness. Several other failures among our 33 ported driver files arose from a violation of the third hypothesis: the complete change was broken up into several steps, and Prequel only returned commits reflecting one step in the series, thus not giving enough information to achieve compatability with the target version. A solution could be to iterate our approach. Finally, our motivating example in Section 2 violates the fourth hypothesis. In addition to the changes in the `suspend` and `resume` fields, the configuration variable `CONFIG_PM` is renamed to `CONFIG_PM_SLEEP`. Although the commit shown in Figure 2 does contain the new configuration variable, the driver maintainer has to be self-motivated to look at it;

gcc does not raise warnings about configuration variables that no longer exist. Tools, such as Undertaker [11], that check for configurability errors could be used in place of gcc within our approach to address this issue.

Our approach provides guidance on how to evolve a driver in concert with other similar drivers, but does not directly address the case where a new kernel feature could better support a unique feature of the device. Our hypothesis, however, is that new features are added to the kernel to support drivers that are already in the kernel, and that the developer adding such a specialized kernel feature will update some of the relevant drivers, if only to validate that the new feature works as expected. The commit logs and change examples identified by Prequel should then still provide guidance on how to apply a new kernel feature in another specialized situation.

7 Related work

The traditional strategy for back porting device drivers involves inserting `#ifdefs` in the driver code to implement different behaviors for different kernel versions. The Linux kernel backports project, initiated in 2007, introduced the use of a compatibility library that abstracts over the variations in different kernels. Rodriguez and Lawall [9] explored the use of Coccinelle to automate the changes needed in a driver to target this compatibility library, and this approach is now actively used by the Linux kernel backports project. These approaches require the developer to manually identify the changes needed for each kernel version, either to modify the driver code directly or to create the compatibility library.

Thung et al. [12] also target automating the backporting of Linux device drivers. Their approach identifies the commit between the source and target versions at which the driver ceases to successfully compile, and then infers transformation rules from the set of changes performed by that commit. While their approach goes further than ours, by inferring transformation rules, it is limited to the information available in the commit that breaks compilation, it has only been evaluated on pairs of successive Linux kernel releases, and it is further limited to drivers in which the compiler signals only one error line. These constraints are not satisfied by many porting issues. For example, for our motivating example in Section 2, there are multiple compiler errors, it was necessary to consult multiple commits, and the commit that breaks compilation does not contain any relevant change examples. Our approach does, however, also assume that only one change is needed for each issue to get from the source version to the target version.

Several recent approaches automate the identification of API evolutions based on analysis of changes in call-graph dependencies [7, 14]. These approaches are well-suited for porting issues that involve only the names of called functions, but not the other types of changes (field type change, etc.) that we have encountered.

Martinez et al. [6] propose a patch query language with the goal of collecting statistics on the frequency of various kinds of code changes to guide automated software repair [13]. Their approach builds on the information about occurrences of a fixed set of change types collected by ChangeDistiller [4]. Change types refer to various syntactic categories, such as removal of an `if` statement, but do not contain information about concrete terms such as function names or structure fields, as we require to limit the results to the commits relevant to a given porting problem. The approach furthermore focuses solely on changes, and thus does not allow queries on context code, as we have also found essential. Stevens et al. [10] propose a query language for changes identified by ChangeDistiller, relying on a logic-programming-based notation. They use their approach for studying instances of refactorings, rather than porting problems.

8 Conclusion

Porting device drivers is an ever-present problem in the context of the Linux kernel. A major challenge in the porting process is to obtain adequate information as to how the port should be carried out. Indeed, the Linux kernel interface is huge, and many relevant details about an interface change are only known to the specific maintainer who has carried it out.

In this paper we have proposed an approach to extract information from compiler output and a git commit history about where changes are needed and how to carry those changes out. On 33 driver files, exhibiting 107 porting issues, our approach enabled us to address 3/4 of the issues, with no specific knowledge about the drivers concerned. Our approach is also reasonably efficient, producing complete results for a driver in at most a few minutes for many cases.

Future work will involve improving performance and addressing the identified limitations, such as the requirement of only one change per issue between the source and target versions. Inferring changes automatically from examples would then be the next major step.

Acknowledgments. We thank the anonymous reviewers and our shepherd Daniel Williams for their feedback on the paper. This work is supported in part by OSADL and by ANR ITrans.

Availability. Our tools and the 33 driver file experiments are available at http://prequel-pql.gforge.inria.fr/

References

[1] APPEL, A. W. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] BRUNEL, J., DOLIGEZ, D., HANSEN, R. R., LAWALL, J. L., AND MULLER, G. A foundation for flow-based program matching: using temporal logic and model checking. In *POPL* (2009), pp. 114–126.

[3] Curl. https://curl.haxx.se/.

[4] FLURI, B., AND GALL, H. C. Classifying change types for qualifying change couplings. In *14th IEEE International Conference on Program Comprehension* (2006), pp. 35–45.

[5] LAWALL, J., LAMBERT, Q., AND MULLER, G. Prequel: A patch-like query language for commit history search. Research Report RR-8918, Inria Paris, June 2016.

[6] MARTINEZ, M., DUCHIEN, L., AND MONPERRUS, M. Automatically extracting instances of code change patterns with AST analysis. In *ICSM* (2013), pp. 388–391.

[7] MENG, S., WANG, X., ZHANG, L., AND MEI, H. A history-based matching approach to identification of framework evolution. In *ICSE* (2012), pp. 353–363.

[8] PADIOLEAU, Y., LAWALL, J. L., HANSEN, R. R., AND MULLER, G. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys* (2008), pp. 247–260.

[9] RODRIGUEZ, L. R., AND LAWALL, J. Increasing automation in the backporting of Linux drivers using Coccinelle. In *11th European Dependable Computing Conference - Dependability in Practice (EDCC)* (2015), pp. 132–143.

[10] STEVENS, R., AND ROOVER, C. D. Extracting executable transformations from distilled code changes. In *Software Analysis, Evolution, and Reengineering (SANER)* (2017), pp. 171–181.

[11] TARTLER, R., LOHMANN, D., SINCERO, J., AND SCHRÖDER-PREIKSCHAT, W. Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem. In *EuroSys* (2011), pp. 47–60.

[12] THUNG, F., BACH, L. D. X., LO, D., AND LAWALL, J. Recommending code changes for automatic backporting of Linux device drivers. In *ICSME* (2016), pp. 222–232.

[13] WEIMER, W., NGUYEN, T., LE GOUES, C., AND FORREST, S. Automatically finding patches using genetic programming. In *ICSE* (2009), pp. 364–374.

[14] WU, W., GUÉHÉNEUC, Y.-G., ANTONIOL, G., AND KIM, M. AURA: a hybrid approach to identify framework evolution. In *ICSE-Volume 1* (2010), pp. 325–334.