META

- Estimated time of presentation: 17~20 minutes, leaving 5~8 minutes for Q & A.
- Don't speak too fast. Slow down.

HELLO

Thank you for the kind introduction. Good morning, everyone. I am honored to present our paper "Is it Possible to Automatically Port Kernel Modules". I'm the third author, Zhenyang Dai from Tsinghua university. This is a work from Yanjie Zhen, Wei Zhang, Junjie Mao, Yu Chen and me.

NEW PAGE

BACKGROUND

Our paper, as its title suggests, contains an empirical study on the characteristics of kernel internal interface changes, and analyzes the feasibility and possible challenges in automatically porting kernel modules.

NEW PAGE

Kernel modules, as a functionality of the Linux kernel, provide a flexible way to extend the ability of the kernel. Nowadays, they are an essential part of the kernel, accounting for over 70% of the Linux kernel, as you can see in the figure.

NEW PAGE

Porting kernel modules is necessary while also difficult. It's necessary, because the kernel APIs used by kmods are fast evolving and not stable. The linux kernel does not want compatibility to become a burden. So, with unstable kernel interface, kernel modules working with one versi.n of the kernel generally fail to work with another. In case you don't know, porting is the process of updating the kmods to let them work with different versions of the kernel, analogous to porting python 2 code to python 3 code. And there are reasons that kmods should run on different version of the kernel. In the case of forward porting, kernel modules work with newer kernels and thus enjoy better security, performance and new functionalities. And sometimes people may want a kmod to work with older kernels, for stability or compatibility reasons. However, porting kmods is not an easy job. The Linux kernel is a large and complex system involving many topics ranging from underlying hardware to complexities in the C language. And it's growing and evolving rapidly and continuously.

NEW PAGE

Currently, porting kmods is mainly done manually by developers using a trial-error-fix approach. Developers first compile their kmods against the version of kernel they wish to port the kmod to, then inspect and understand compiler errors and warnings by finding the patches that changed some kernel interface and caused the errors or warnings, and then modify the kmods until they get no error or warnings. Obviously, this process is tedious and error-prone, mainly because of the large number of patches, for example there are over 480 000 patches from Linux version 3.0 to version 4.16. What's worse, with such massive information, kmod developers are likely to get distracted by irrelevant details.

While porting kmods manually remains the common way, many automated tools have been built in the last decade. The tools mainly fall into two categories. Tools of the first category employ automated program repair techniques, just like the trial-error-fix approach, which you can call the dont-change-compile-get-error-and-warnings-do-change-until-no-error-and-warning approach. They rely heavily on compiler errors and warnings. Tools of the second category use instance-based inference techniques. They extract change patterns from in-tree kmods which are taken care of by the kernel developers. But most pattern inference tools are not pragmatic, since the kernel and kmods use C language features such as macros, callback function pointers etc, which hinder accurate analysis on inferring change patterns.

NEW PAGE

STAY FOR 4 SECONDS

NEW PAGE

INTRODUCTION

Our paper consists of an empirical study that we conducted on 200 commits involving 10 active kmods in the past 7 years. Recall that kernel API changes cause porting kmods to be necessary, hence we focus on the characteristics and representation of kernel API changes, especially those connected with automatic kmod porting. We further investigate the feasibility and possible challenges in automatically porting kmods, and provide some guidance on designing automated tools. To avoid misunderstanding, the scope the this paper does not include how to build an automated tool. We just discuss few things worth of notice when trying to build one.

NEW PAGE

Our study extensively use the concept of change patterns, or just patterns for short so I'll explain it here. By change pattern, we mean some abstract modification to the source code. They describe how usage of some kernel API should be updated when the kernel API is changed in some way.

Let's put it this way. In the Linux kernel, a commit modified some kernel API. As a result, a kmod using that kernel API needs to be updated. Now we have a list of places in the kmod that will be updated. We call each place as a change instance. Change pattern means how we would modify a piece of code that uses the changed kernel API. You can imagine that if we see enough change instances, we can infer the underlying change pattern even without knowing how the kernel API changed exactly.

Change patterns are important because they can help automatic porting. There are existing tools that can apply change patterns to kmods automatically. Thus having change patterns means we can update kernel API usages automatically. That's a major proportion of the work in porting kmods.

Also I'd like point out that one commit may contain multiple patterns, and a pattern often consists of multiple instances.

In our experiment, all the patterns we use are inferred manually. And to describe change patterns, we use the semantic patch language SmPL.

NEW PAGE

To make it clear, let's look at an example. This is part of a commit in the Linux repository, and we made a few modifications for reading easier. On the left are the actual changes of the source code.

There are three functions, f2, f3 and f4. They are the f2 callback field in the structs ops1, ops2 and ops3 respectively. This patch adds to them a parameter i of type `inode` pointer, and it replaces all usages of `d_inode(d)`. Also, it introduces a field access wrapper for the `d_sb` field of `dentry`.

Let's try to extract change patterns manually from the left hand side. Well, we can find two patterns. Our description of the first pattern could be like the one on the right. f is a function, just like f2, f3 and f4 on the left. It satisfy certain conditions, which we refer to as contexts. We modify f in some way. For the second pattern we can also describe it in natural language similarly. So here we get two change patterns. And there are three and two change instances for them respectively.

NEW PAGE

Now we use semantic patch language instead of natural language to describe the patterns. On the right are the change patterns in SmPL. The first four lines require that f acts as a callback f2 in a struct ops. Such is a "callback context". Then from line 5 to line 16, we specify the first pattern. That's adding an argument with type inode pointer, and it replaces all usages of `d_inode(arg1)`. The last 6 lines specify the second pattern, a field access wrapper. **PAUSE FOR 2 SECONDS**

I hope what I said just now helps you understand the concept of change patterns. We'll use it shortly.

NEW PAGE Now you know the notion of change patterns, we move on to the experiment and results part.

EXPERIMENT & RESULTS

We choose 10 active kmod that are in the kernel source tree. These kmods are maintained by kernel developers. Then we fetch all patches that modify these kmods and are related to API changes from the last 7 years and manually filter out non-essential patches such as typo fixes or updating comments. After this we get a total of 1753 distinct patches. In the table you see 2081. It's because a patch could change multiple kmods and is therefore counted multiple times. Due to limited time we randomly choose 200 of them for analysis. From the 200 patches we infer 407 patterns manually.

NEW PAGE

Finally we come to the results, which we show through a series of questions. Prior methods based on pattern inference often make assumptions on inputs. By inputs I mean commits, or patches. But are these assumptions really valid in real world?

Some methods take API changes done manually as input and infer patterns by abstracting away irrelevant details. They are only applicable when the input contains only one pattern. Also, we care about the number of instances of each pattern, as it's easier to infer accurate patterns from more instances. In fact, pattern inference only makes sense when there are multiple instances. With only one instance, the concept of pattern is quite meaningless. The last question relates to the methods leveraging compiler messages to point out incorrect API usages.

NEW PAGE

For the first question, our result is shown in the histogram. It displays the distribution of patterns per commit, as well as the proportion of commits involving modification irrelevant to kernel API updates. **PAUSE FOR 3 SECONDS**.

36.5% of the commits contain multiple patterns in a single commit. Furthermore, the shaded part, that's 55.5% of commits involve changes irrelevant to API usage updates, meaning that automated tools should be able to deal with this kind of distraction.

NEW PAGE

Now for the second question, number of instances per pattern, see the distribution on the right.

The majority of patterns, 87.5%, have multiple instances. We said that the more instances per pattern, the easier it is to infer the patterns. Consequently, pattern inference could be useful in most circumstances.

NEW PAGE

For the third question, compiler messages, the figure shows the kinds of results that would be caused if a pattern is missed. By missing a pattern, we mean failure to update the change instances of it as if we forgot about it when porting a kmod. **PAUSE FOR 3 SECONDS**

Although more than half of patterns when missed lead to compiler complaints, that's the first two columns error and warning, quite a few patterns, when missed, do not trigger any compiler error / warning. Worse still, 12.5% of the patterns, when missed, cause no compiler complaints but change program run-time behavior, that's the third column runtime. Even for patterns that do not change run-time behavior of kmods, it's possible that they involve API usages soon to be deprecated. In fact according to the commit messages, 35 of them, as in the future column, are indeed so.

NEW PAGE

Now we move on to another point. It's important that we represent changes in a commit effectively in a structured way, in a way that helps inference of patterns. We point out some important information that has to be preserved in such representation, by answering the following questions. By the way, as it is obviously impossible or very unreliable to infer patterns from a single instance, in this section we limit our dataset to the 356 patterns that have multiple instances.

NEW PAGE

First, we need to understand what kinds of code are most often touched by the change patterns. On the right are kinds of code and the frequency of such code being modified by some pattern. **PAUSE FOR 3 SECONDS**

From the figure, we conclude that the majority of patterns, say 86.5% the first two columns, touch function definitions, and very rarely do they need to modify multiple functions simultaneously, that's the second column. 6.4% of patterns update global variables, as in the global column. Mostly that means adding or deleting field initializers to or from structs. Although the patterns update other kinds of code, these two account for the vast majority. Thus, intra-procedural analysis would suffices to infer most patterns.

NEW PAGE

Second, as the Linux kernel is written in the C language, we need to deal with the complexities of it. Most importantly, we have to deal with macro invocation. If we fail to do so, it's possible that our tool could only work with preprocessed C code without macros.

The figure shows the number of patterns containing macro invocations in the deleted or inserted lines. The notion of deleted and inserted lines come from SmPL. **GO TO NEXT PAGE** As you see on the right, green lines are inserted and red lines are deleted. **RETURN**

We see that 33.4% of the patterns we look at involve macro invocation. So designers of tools for automatic porting kmods have to take macros into consideration.

NEW PAGE

NEW PAGE

Third, we investigate the contexts of the patterns. **GO TO PREVIOUS PAGE** In our example, we already introduced this concept. See the first 4 lines on the right. It's a callback context. **RETURN**. There are other contexts, such as global variables, which also mostly means updating field initializers in structs. Our results show that 36.5% of patterns require some context. Among them, 78.5% require either the callback context or the global variable context, just see the callback column and global column.

NEW PAGE

DISCUSSION

Based on the above results, we move on to the discussion of feasibility and challenges of automatic kmod porting.

NEW PAGE

First we discuss feasibility. As have been discussed, the two main ways to automatic port kmods are based on compilation messages and pattern inference. Our study suggests that, with 38.4% of API changes not introducing any compiler messages, the former approach is severely limited. However, recall that the more instances per pattern, the easier pattern inference is. With 87.5% of patterns having multiple instances, the latter approach is promising.

Now we look at the challenges. The first one is effective pattern extraction in the presence of noises. When inferring patterns, automatic porting tools have to be able to cope with noises in the inputs, for example multiple patterns tangled in a single commit or code changes irrelevant to API usage updates. Secondly they have to work with macros. The most straightforward way is to infer patterns containing macros. Lastly, automated tools must infer not only patterns but also the contexts of the patterns, especially callbacks and global variables. Unfortunately, existing approaches do not perform very well in dealing with these issues.

NEW PAGE

GOODBYE

That's all my presentation on "Is It Possible to Automatically Port Kernel Modules", an empirical study the characteristics of kernel internal interface changes. If you are interested, please check our paper.

Thank you very much for listening. And now, and questions?