Understanding the Genetic Makeup of Linux Device Drivers

Peter Senna Tschudin[1] Laurent Réveillère[2] Lingxiao Jiang[3] David Lo[3]
Julia Lawall[1] Gilles Muller[1]

[1]LIP6 – Inria & UPMC, [2]LaBRI, [3]Singapore Management University

Peter.Senna@lip6.fr, Laurent.Reveillere@labri.fr, {lxjiang,davidlo}@smu.edu.sg, {Julia.Lawall,Gilles.Muller}@lip6.fr

Abstract

Attempts have been made to understand driver development in terms of code clones. In this paper, we propose an alternate view, based on the metaphor of a gene. Guided by this metaphor, we study the structure of Linux 3.10 ethernet platform driver probe functions.

1. Introduction

Over the past 15 years, substantial attention has been paid in the operating system (OS) community to the problem of developing reliable device drivers. Proposed approaches range from the use of domain-specific languages [12], to the use of templates [2], to the generation of provably correct drivers from specifications [15]. Nevertheless, Kadav and Swift have pointed out that many of these approaches rely on assumptions that are not true for many classes of drivers, and thus do not scale up to the complete range of devices [6]. There is thus a need for a better understanding of the structure of device drivers, and of the specific issues and design decisions that go into their creation.

A common view of device driver development is that drivers are implemented by copy-paste programming. In this form of programming, a driver developer starts with driver code for a related device, makes a copy, and then adjusts the copy according to device-specific information. In order to understand this phenomenon, a number of studies have applied code clone detectors to Linux code [6, 10, 18]. However, knowing the amount of duplicated code, as often reported in research papers, or the set of fragments of duplicated code, as reported by code clone detection tools, is not very helpful in understanding how to create a driver. Instead, what is needed is a way to justify which fragments of an existing driver should be selected, to achieve what functionality.

Our thesis is that the set of code fragments included in a device driver is determined by the set of features provided by the device and the target machine, and by developer choices in terms of the use of OS and support libraries. Furthermore, we posit that these features are reflected by the data structures manipulated by the driver. Put another way, we view the code fragments that make up a driver as being drawn from a set of *genes* that support the various possible features of a device. A gene is a sequence of related, possibly noncontiguous code fragments that realize the functionality relevant to a feature. We view the construction of a driver as the process of selecting, adapting, and, if needed, interleaving these genes to achieve a desired effect. Thus, when a driver developer copies some existing code, he replicates some genes and deletes others, that do not correspond to the features of the target device.

In this paper, we perform a preliminary investigation of this thesis. For this, we manually identify a part of the driver genome, and then study the dispersal of these genes within a driver population. In the context of this preliminary work, our experiments focus on the `probe` functions of Linux ethernet platform drivers from Linux 3.10, released in June 2013. These functions are concerned with driver initialization. They are interesting for this work because they are affected by a wide range of device and machine features and because they involve OS and support libraries that have evolved over time. Our study reveals the following:

- The ethernet probe entry points use common sequences of operations that can be viewed as genes.
- A single gene may involve functions defined at multiple levels of the OS, ranging from the core to various support libraries.
- New gene variants have appeared over time, resulting multiple coexisting ways to express the same behavior.
- There is some correspondence between the choice of genes and the data structures manipulated by probe entry points, but this view needs some refinement to account for derived values.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLOS'13, November 03 - 06 2013, Farmington, PA, USA.
Copyright © 2013 ACM 978-1-4503-2460-1/13/11...$15.00.
http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2525528.2525536

The rest of this paper is organized as follows. Section 2 illustrates a typical probe function. Section 3 presents the design of our experiments, and the variant of an existing clone detection tool that we have developed to help identify genes in Linux device driver code. Section 4 presents the results of our experiments. Section 5 describes related work and Section 6 concludes.

2. Linux Platform Driver Probe Functions

To limit the amount of code to study in this preliminary work, we consider only the probe function entry points of Linux ethernet platform drivers, *i.e.*, the functions mentioned in the `probe` field of a `platform_driver` structure of an ethernet driver found in the Linux kernel directory `drivers/net/ethernet`. A platform driver is a driver for a device that offers direct addressing from the CPU, instead of being attached to a bus such as PCI or USB. In Linux 3.10, over 1300 Linux driver files for all types of services implement the platform driver interface. Of these, 65, amounting to 77,171 lines of code,[1] are ethernet drivers found in the directory `drivers/net/ethernet`.[2] The `probe` entry point is invoked when the device is initialized, to perform any device-specific initializations. The considered Linux 3.10 ethernet probe entry points amount to 5,838 lines of code. Roughly a third of these probe functions (20) are from drivers for the Arm architecture, and about a sixth (12) are for PowerPC.

Figure 1 shows a typical probe entry point. This function performs three basic operations, which we recognize as genes: 1) initializing memory mapped I/O (in red, lines 6-7, 13-16, 25, 27), 2) initializing interrupts (in blue, lines 8-9, 17), 3) connecting the device to the net device support library (in green, lines 10-11, 19-20, 31). Note that the code for initializing memory mapped I/O contains a bug; line 27 should call `release_mem_region` rather than `release_resource`. Indeed, the possibility of bugs in the code increases the difficulty of identifying genes automatically, and is one reason why clone detection is not sufficient.

In general, a probe entry point may call other functions that typically perform device-specific initializations. For simplicity, in this preliminary work, we do not take these functions into account.

3. Studying the Driver Genome

To evaluate our thesis, we carry out a study in two parts. First, to find genes, we pass over the set of probe entry point functions, to identify common sequences of possibly disjoint code fragments, which we abstract into genes. Then, to study gene dispersal, we pass over these functions again, now with the gene definitions, and count the number of occurrences

[1] Lines of code were calculated using SLOCCount.

[2] We observe that the Linux directory `drivers/net/ethernet` contains drivers that are not ethernet drivers, but are drivers for related services, such as the MDIO bus. We have not included these drivers in our study.

```
1  static int ftmac100_probe(struct platform_device *pdev) {
2    struct resource *res; int irq, err;
3    struct net_device *netdev; struct ftmac100 *priv;
4
5    if (!pdev) return -ENODEV;
6    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
7    if (!res) return -ENXIO;
8    irq = platform_get_irq(pdev, 0);
9    if (irq < 0) return irq;
10   netdev = alloc_etherdev(sizeof(*priv));
11   if (!netdev) { ... goto err_alloc_etherdev; }
12   ...
13   priv->res = request_mem_region(res->start, resource_size(res), ...);
14   if (!priv->res) { ... goto err_req_mem; }
15   priv->base = ioremap(res->start, resource_size(res));
16   if (!priv->base) { ... goto err_ioremap; }
17   priv->irq = irq;
18   ...
19   err = register_netdev(netdev);
20   if (err) { ... goto err_register_netdev; }
21   ...
22   return 0;
23
24 err_register_netdev:
25   iounmap(priv->base);
26 err_ioremap:
27   release_resource(priv->res);
28 err_req_mem:
29   netif_napi_del(&priv->napi);
30   platform_set_drvdata(pdev, NULL);
31   free_netdev(netdev);
32 err_alloc_etherdev:
33   return err;
34 }
```

Figure 1. faraday/ftmac100.c probe entry point

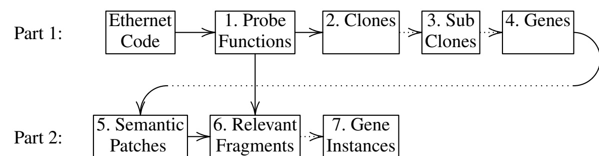

Figure 2. Methodology: solid lines represent automatic tools, while dotted lines represent manual analysis

of each gene. These steps are summarized in Figure 2. Currently, our methodology is primarily manual, represented by dotted lines in Figure 2, with some preliminary tool support.

The steps in our methodology are as follows, starting from a set of Linux source code files. In Figure 2, the result of step n is labeled n.

Part 1: Gene identification

1. Identify and extract the `probe` entry points. For this, we use Coccinelle [14], to find declarations of `platform_driver` structures, and extract the definition of the function stored in the structure's `probe` field.
2. Identify clones in the probe entry point code. Our assumption is that genes contain code that is common to

a number of drivers. We thus use clone detection to highlight likely relevant code, to ease the identification of genes. For clone detection, we use Deckard[3] [5], which can find clones at any degree of granularity. We put no minimum on the clone size and require clones to be exact matches. Nevertheless, we have substantially changed the strategy of Deckard for abstracting over subterms, mitigating the effect of requiring exact matches. In particular, we have modified Deckard to distinguish function names, structure field names, and case labels, which appears to be essential for accurately treating systems code (*cf.* [6]). On the other hand, we abstract over debugging code and error-handling code, which tend to be driver-specific.

3. Subdivide clones, as needed, according to their functionality. Clone detectors report maximal recurring code patterns, regardless of their semantics. Thus, different parts of a clone may relate to different genes.
4. Manually collect the identified code fragments that realize a given functionality into genes. We typically identify a gene as a sequence of code fragments involving common variables, but ad hoc knowledge of the semantics of the OS and OS support functions is also required.

Part 2: Gene dispersal evaluation

5. Create Coccinelle semantic patches (pattern matching specifications) that detect occurrences of the identified genes. These semantic patches are designed to identify exact occurrences of the genes, as well as fragments of code that may relate to a gene, but do not match the gene exactly. This helps address the issue of bugs in gene instances, as highlighted in Section 2.
6. Apply the semantic patches to the probe functions.
7. Manually check and count gene instances among the matched code fragments.

4. Evaluation

We have focused our evaluation on the genes illustrated by our example in Figure 1. These are related to 1) allocation of addresses for use with memory mapped I/O, 2) allocation of interrupts, and 3) driver registration. Other genes that we have identified, but have not yet studied in detail, relate to DMA, clocks, MAC addresses, locks, timers and delays.

4.1 Memory mapped I/O

The genes for initializing memory mapped I/O each: 1) obtain the physical addresses associated with the device, 2) reserve these addresses, and 3) map these addresses into virtual ones. There are two families of these genes, depending on how the physical address information is obtained.

***Memory mapped I/O based on* `platform_get_resource`.** One family of genes, illustrated by the paths in the directed graphs shown in Figure 3, obtains the range of physical addresses using the function `platform_get_resource` with the flag `IORESOURCE_MEM` as an argument. For conciseness, the graphs in Figure 3 describe the genes in terms of only the names of the principal functions called, and not any possible assignments, tests, and error-handling code. Cleanup code, to be performed only in case an error is detected in the overall probe process is shown in brackets. The variant consisting of the left most path in the leftmost graph is the one found in our example in Figure 1.

[3] https://github.com/skyhover/Deckard

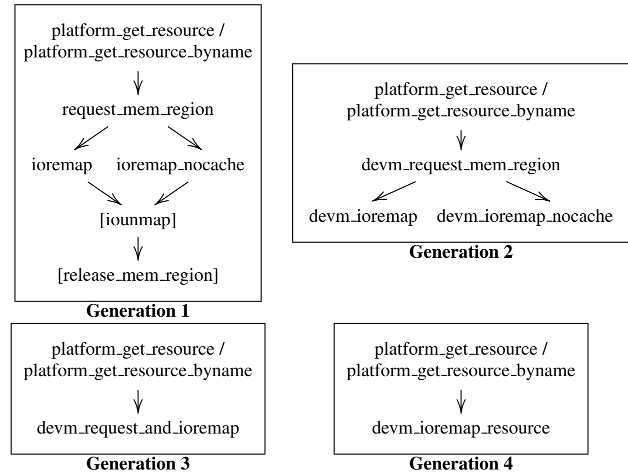

Figure 3. Evolution in the genome for mapping virtual memory

The upper leftmost graph in Figure 3 represents the original genome. The developer can choose between `platform_get_resource` and `platform_get_resource_byname` depending on the kind of key to use to find the device information, and between `ioremap` and `ioremap_nocache` depending on whether caching and buffering should be used for I/O with the device. Finally, `request_mem_region` is not always present. If it is omitted, the developer must have confidence that the OS will not use the region of physical memory for another purpose, and so it is not necessary to use `request_mem_region` to reserve it. The remaining graphs show subsequent extensions to the genome. These extensions have been driven by the goal of simplifying driver code and reducing the likelihood of errors.

Figure 4 summarizes the number of probe entry points that use at least one instance of each of these genes. In a few cases, a driver tries both `platform_get_resource` and `platform_get_resource_byname`, but the rest of the code is shared; we count these as two separate gene occurrences. Overall, the gene for memory mapped I/O that is used in Figure 1 is the most common, appearing in 15 files. Most of the other genes only appear in at most one or two probe entry points. Clone detection using any kind of minimum threshold on the number of occurrences of a clone would

Generation 1:

	request ioremap	only ioremap	request ioremap_nocache	only ioremap_nocache
pgr	15	4	2	2
pgr_byname	1	0	1	1

Generation 2:

	request ioremap	only ioremap	request ioremap_nocache	only ioremap_nocache
pgr	2	1	2	0
pgr_byname	0	0	0	0

Generations 3 and 4:

	devm_request_and_ioremap	devm_ioremap_resource
pgr	2	2
pgr_byname	0	0

Figure 4. Number of probe entry points containing the platform_get_resource-based genes for initializing memory-mapped I/O. "pgr" abbreviates "platform_get_resource".

thus likely not be sufficient to find the relationship between these drivers.

Five probe entry points use `platform_get_resource` or `platform_get_resource_byname`, but do not match any of our genes. In two cases, the rest of the gene is found in a helper function, which is out of scope of our current study. In one case, the obtained address is only used in a debugging message. In the remaining cases, it may be that the missing `ioremap` is known not to be needed.

Finally, we observe that these genes cross kernel libraries. `platform_get_resource` and `platform_get_resource_byname` are specific to platform drivers. `request_mem_region`, `ioremap`, and `ioremap_nocache` are generic I/O operations. Finally, the "devm" variants are part of the devres library.

***Memory mapped I/O based on* `of_address_to_resource`.** A few drivers, targeting the PowerPC architecture, obtain the physical address range using the function `of_address_to_resource`. These may then use *e.g.*, `request_mem_region` and `ioremap`, as in the `platform_get_resource` case, or they may use only the function `of_iomap`, which encapsulates both a call to `of_address_to_resource` and a call to `ioremap` (the physical address range is not reserved in these cases). A final variant is `of_ioremap`, which is not used with `of_address_to_resource`, and which either provides a SPARC-specific definition of `ioremap` or is a wrapper for `request_mem_region`, for a 32 or 64-bit architecture, respectively. Three probe entry points explicitly use `of_address_to_resource` and `ioremap`, of which two also use `request_mem_region`. Three probe entry points use `of_iomap` alone, while two use `of_ioremap`.

Connection with data structures. Part of our thesis is that the gene instances found in a driver are determined in part by the data structures manipulated by the driver. In the case of memory mapped I/O, the location that receives the starting virtual memory address has type `void *`, which is not specific to this kind of gene. Nevertheless, out of the 53 initializations of memory mapped I/O found in our probe entry points, 89% (47) are stored in a field of the driver's private structure that is annotated with `__iomem`. Of the remainder, 4 are stored in a field that is not annotated, 1 is only via a local `__iomem` variable within the probe entry point, and 1 is used via a global `__iomem` variable.

On the other hand, not every field annotated with `__iomem` implies a separate instance of one of our genes. In some cases, multiple `__iomem` fields do imply multiple gene instances. In other cases, however, these fields are used to store addresses at an offset from a value computed using a gene instance. We leave further investigation of these cases to future work.

	pgr	pgr_byname	pgi	pgi_byname
alone	15	0	20	9
with request_irq	2	0	8	0

Figure 5. Number of probe entry points containing the platform_get_resource-based genes for obtaining IRQs. "pg{r,i}" abbreviates "platform_get_{resource,irq}".

4.2 Interrupts

As for physical address ranges, to be used for memory mapped I/O, interrupt numbers may be obtained using `platform_get_resource` or `platform_get_resource_byname`, this time with the flag `IORESOURCE_IRQ`. The driver then dereferences the returned resource's `start` and `end` fields to obtain the range of interrupts. A simplified form exists as well; the functions `platform_get_irq` and `platform_get_irq_byname` both obtain the resource structure and return the value in the `start` field, and are useful when only the first interrupt number within the range is needed.

Obtaining the interrupt number, however, is not sufficient to initiate interrupt handling. It is also necessary to invoke the function `request_irq`. Typically, this is done in another driver entry point, but some drivers call this function in the probe function. We thus find that a single gene can affect multiple entry points, and the effect of a gene can be distributed across the diverse entry points in different ways.

Figure 5 shows the number of probe entry points using the various genes. We see that the use of the two variants `platform_get_resource` and `platform_get_irq` is roughly even, while the "byname" variants are used rarely. Furthermore, there are few probe entry points that call `request_irq`.

4.3 Registering a network device

In order to register a network device with the kernel, a network driver creates a `net_device` structure, initializes its fields, and passes the resulting structure to the function `register_netdev`. In the case of an ethernet driver, three functions are available to allocate a `net_device` structure: `alloc_etherdev`, `alloc_etherdev_mq`, and `alloc_etherdev_mqs` differing only in the number of transmit and

receive queues that are allocated. This leads to three basic genes. 76% (50) of our probe entry points use `alloc_etherdev`, 4% (3) use `alloc_netdev_mq`, and 1% (1) use `alloc_netdev_mqs`. An additional 16% (11) invoke one of these functions via a help function.

The three basic genes come in a number of variants, depending on which fields of the `net_device` structure are additionally initialized. In this case, we consider the initialization to be part of the device registration gene, but the computation of these values to be separate, whether part of another gene or device-specific code. Fields that can be initialized include the field `base_addr` (19 files), to be initialized to the physical address to be used for I/O (*cf.*, Section 4.1), and the field `irq` (33 files) to be initialized with the device IRQ number (*cf.*, Section 4.2).

Unlike the other considered genes, the main goal of the device registration gene is not to initialize a field of the device's private structure. Nevertheless, we can view the inclusion of the device registration gene as being motivated by the presence of a local variable of type `net_device` in the definition of the probe entry point.

5. Related Work

Clones. Wang and Godfrey investigate clones in Linux SCSI drivers. On Linux 2.6.32.15, they find clone rates of 10-18% depending on the kind of code considered. They furthermore show that a high rate of clones between two drivers implies that the drivers support the same set of busses, with a success rate of over 80%. This result is consistent with our thesis that driver code is determined by device features.

Antoniol *et al.* [1] studied clones in various Linux 2.4 versions. They found few clones across different subsystems, *e.g.*, arch vs. drivers, but they found clone rates of up to 22.61% between different architectures. The case of different architectures, may be considered to be analogous to the case of different devices, where the code to address specific features may be in common.

Several clone detection tools have been specifically designed to address the properties of systems code. CP-Miner [10], based on the identification of common subsequences within basic blocks, finds that over 21% of Linux 2.6.6 driver code is covered by clones. DrComp [6] works at the function level, and finds similarities within 8% of driver code, in Linux 2.6.37.6. These tools identify the presence of clones, but do not explain how this cloned code contributes to the driver structure.

In practice, we have found that the code fragments that make up a single gene can be widely separated within a driver. A number of works provide mechanisms to identify so-called API usage protocols, which are commonly occurring not-necessarily-contiguous sequences of operations [4, 8, 9, 11, 13]. The identified protocols have then been used to construct fault finding rules. These approaches, however, have do not try to exhaustively infer the protocols relevant to a code base nor do they consider compositions of protocols as a means of describing code structure.

Driver structure. Spear *et al.* [17] propose a notation for declaring the resource requirements of a device driver, in the context of the OS Singularity. Their declarations are primarily intended for verification, but also trigger the generation of boilerplate code targeting a high-level resource acquisition interface. While this work shares with our work a focus on resources, it does not target legacy interfaces, provide an explanation of legacy code, or address the interleaving of resource acquisitions and device-specific operations.

Termite proposes the derivation of a driver implementation from specifications of the OS, the device, and the device class [15]. Termite specifications are concerned with input-output behavior, *e.g.*, the list of OS entry points called by the OS and the types of results they are supposed to produce, and does not address the kinds of code that are common across driver implementations. RevNic proposes to port drivers based on templates and device-specific values obtained from executing a driver for the same device but a different OS [2]. Templates must be written manually. The fact that we find that some genes, *e.g.* for initializing memory mapped I/O, occur different numbers of times depending on device needs suggests that a single fixed template may be insufficient.

Feature-oriented programming is a programming paradigm in which a complete *software product line* is described at once, and specific programs are generated by selecting an appropriate set of features from a formally defined feature model. Beuche *et al.* organize an OS as a product line in order to be able to generate minimal, application-specific instances. Later work has focused on configuration. She *et al.* transform a set of unstructured configuration options into a feature model [16]. Dietrich *et al.* take makefile information into account [3]. These approaches organize existing feature decompositions, rather than identifying features in the code. Kästner *et al.* mine feature implementations in existing Java code, based on a preliminary manual identification of features and some relevant code fragments [7]. This work could be useful to us in the future, for identifying gene implementations automatically. Our current focus is on understanding what features and feature implementations make up device driver code.

6. Conclusion

In this paper, we have identified a new way of understanding device driver structure, in terms of the genes that are used in their construction, and identified the connection between the presence of these genes and the set of data structures manipulated by the driver. In this preliminary work, we have focused on the portions of these genes found within Linux ethernet platform driver probe functions. In the more general case, a gene covers all operations related to a given feature, *e.g.* interrupt management, and can have an impact on any

of the operations performed by a driver. In the future, we will expand the genome, increase the degree of automation of our methodology, consider how to identify and specify dependencies between genes, and, ideally, propose a means of automatically generating a device-specific skeleton of a driver implementation from minimal information about the features and requirements of the device.

References

[1] ANTONIOL, G., VILLANO, U., MERLO, E., AND PENTA, M. D. Analyzing cloning evolution in the Linux kernel. *Information and Software Technology 44*, 13 (2002), 755–765.

[2] CHIPOUNOV, V., AND CANDEA, G. Reverse engineering of binary device drivers with RevNIC. In *EuroSys* (Paris, France, Apr. 2010), pp. 167–180.

[3] DIETRICH, C., TARTLER, R., SCHRÖDER-PREIKSCHAT, W., AND LOHMANN, D. A robust approach for variability extraction from the Linux build system. In *16th International Software Product Line Conference (SPLC)* (Salvador, Brazil, 2012), vol. 1, pp. 21–30.

[4] ENGLER, D. R., CHEN, D. Y., CHOU, A., AND CHELF, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP* (Banff, Canada, Oct. 2001), pp. 57–72.

[5] JIANG, L., MISHERGHI, G., SU, Z., AND GLONDU, S. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE* (Minneapolis, MN, USA, May 2007), pp. 96–105.

[6] KADAV, A., AND SWIFT, M. M. Understanding modern device drivers. In *ASPLOS* (London, UK, Mar. 2012), pp. 87–98.

[7] KÄSTNER, C., DREILING, A., AND OSTERMANN, K. Variability mining: Consistent semiautomatic detection of product-line features. *TSE*. to appear.

[8] LAWALL, J. L., BRUNEL, J., PALIX, N., HANSEN, R. R., STUART, H., AND MULLER, G. WYSIWIB: exploiting fine-grained program structure in a scriptable API-usage protocol-finding process. *Software: Practice and Experience 43*, 1 (2013), 67–92.

[9] LE GOUES, C., AND WEIMER, W. Specification mining with few false positives. In *TACAS* (York, UK, Mar. 2009), vol. 5505 of *LNCS*, pp. 292–306.

[10] LI, Z., LU, S., MYAGMAR, S., AND ZHOU, Y. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI* (Dec. 2004), pp. 289–302.

[11] LI, Z., AND ZHOU, Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE* (Lisbon, Portugal, Sept. 2005), pp. 306–315.

[12] MÉRILLON, F., RÉVEILLÈRE, L., CONSEL, C., MARLET, R., AND MULLER, G. Devil: An IDL for hardware programming. In *OSDI* (San Diego, CA, Oct. 2000), pp. 17–30.

[13] NGUYEN, T. T., NGUYEN, H. A., PHAM, N. H., AL-KOFAHI, J. M., AND NGUYEN, T. N. Graph-based mining of multiple object usage patterns. In *ESEC-FSE* (Amsterdam, Netherlands, Aug. 2009), pp. 383–392.

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

[15] RYZHYK, L., CHUBB, P., KUZ, I., LE SUEUR, E., AND HEISER, G. Automatic device driver synthesis with Termite. In *SOSP* (Big Sky, MN, USA, Oct. 2009), pp. 73–86.

[16] SHE, S., LOTUFO, R., BERGER, T., WASOWSKI, A., AND CZARNECKI, K. Reverse engineering feature models. In *ICSE* (Honolulu , HI, USA, May 2011), pp. 461–470.

[17] SPEAR, M. F., ROEDER, T., HODSON, O., HUNT, G. C., AND LEVI, S. Solving the starting problem: device drivers as self-describing artifacts. In *EuroSys* (Leuven, Belgium, Apr. 2006), pp. 45–57.

[18] WANG, W., AND GODFREY, M. W. A study of cloning in the Linux SCSI drivers. In *SCAM* (Williamsburg, VA, USA, Sept. 2011), pp. 95–104.