Sound Formal Verification of Linux's USB BP Keyboard Driver

Willem Penninckx, Jan Tobias Mühlberg, Jan Smans, Bart Jacobs, and Frank Piessens

IBBT-DistriNet, KU Leuven, 3001 Leuven, Belgium

Abstract. Case studies on formal software verification can be divided into two categories: while *(i)* unsound approaches may miss errors or report false-positive alarms due to coarse abstractions, *(ii)* sound approaches typically do not handle certain programming constructs like concurrency and/or suffer from scalability issues. This paper presents a case study on successfully verifying the Linux USB BP keyboard driver. Our verification approach is *(a)* sound, *(b)* takes into account dynamic memory allocation, complex API rules and concurrency, and *(c)* is applied on a real kernel driver which was not written with verification in mind. We employ VeriFast, a software verifier based on separation logic. Besides showing that it is possible to verify this device driver, we identify the parts where the verification went smoothly and the parts where the verification approach requires further research to be carried out.

1 Introduction

The safety and security of today's omni-present computer systems critically depends on the reliability of operating systems (OS). Due to their complicated task of managing a system's physical resources, OSs are difficult to develop and to debug. As studies show, most defects causing operating systems to crash are not in the system's kernel but in the large number of OS extensions available [1, 4]. In Windows XP, for example, 85% of reported failures are caused by errors in device drivers [1]. As explained in [4], the situation is similar for Linux and FreeBSD: error rates reported for device drivers are up to seven times higher than error rates stated for the core components of these OSs.

A lot of research aims to prove the correctness of programs. However, not much work has been carried out to test whether the results of this research is applicable to complex real-world programs where correctness is important, like operating systems drivers. To work towards addressing this question, this paper applies a separation-logic based verifier, VeriFast [11], on a device driver taken from the Linux kernel.

The driver code subject to verification is Linux's USB Boot Protocol keyboard driver. While being small, this driver contains a bigger than expected subset of kernel driver complexity. It involves asynchronous callbacks, dynamic allocated memory, synchronization and usage of complex APIs. During verification, we identified and fixed a number of bugs. For these bugs we submitted

patches that have been accepted by the driver's maintainer and are queued for inclusion in future Linux releases.

In the remainder of this paper we briefly introduce VeriFast and the device driver. We outline the verification of the driver and elaborate on the challenges involved. Finally, we discuss related work and draw conclusions.

2 Background

The verifier we apply to the USB BP keyboard driver is VeriFast. VeriFast's underlying logic is based on an extension of separation logic. Separation logic [16] builds on Hoare Logic [10] and adds support for the heap by introducing the separating conjunction $*$ and other assertions describing a heap. An assertion $A * B$ expresses that the heap can be divided in two disjoint parts, such that assertion A holds for the first part and B holds for the second part.

Concurrency is supported by associating a real number (called "fraction") from $(0, 1]$ to every heap cell which is regarded as a permission (e.g. to access data) [3]. Multiple threads can obtain different fractions of the same permission. What is allowed with the permission, depends on the fraction, e.g. for an access-data permission, a fraction 1 denotes read-write permission, a fraction of another size denotes read-only permission.

Specifications for (spin)locks are done in a fashion similar to [7]: with a lock a handle and an invariant are associated. A fraction of the handle allows acquiring the lock, which yields (adds to the thread's owned permissions) the invariant which represents the permissions protected by the lock.

VeriFast checks annotated C files. The annotations can contain pre- and post-conditions written in separation logic, ghost data structures and ghost lemmas. The VeriFast tool and its technical documentation, including a tutorial and a formalization of a core subset of VeriFast and its semantics, are available for download at `http://www.cs.kuleuven.be/~bartj/verifast/`.

3 Overview of How `usbkbd` Works

The driver subject to verification is Linux's USB Boot Protocol keyboard driver, named `usbkbd`[1]. This section gives a high-level overview of how the driver works, leaving out details concerning concurrency and the exact API usage.

On loading, `usbkbd` registers itself with the USB API. When a new keyboard is attached, the API calls the `usb_kbd_probe` function of `usbkbd`. `usb_kbd_probe` checks whether the driver can handle the attached keyboard, and if so initializes a USB Request Block (URB). An URB is an asynchronous request that can be used to send or receive data from a USB device. The purpose of the URB initialized here is to receive key-presses and key-releases. This URB is named the IRQ URB. `usb_kbd_probe` initializes another URB for updating the LED

[1] The driver's source file, `usbkbd.c`, is located in `drivers/hid/usbhid/` in the Linux kernel distribution available from `http://kernel.org/`.

status (e.g. numlock) named the LED URB. `usb_kbd_probe` then registers a new input device with the input API to make the keyboard available to applications. When the newly created input device is opened, `usbkbd`'s `usb_kbd_open` callback is invoked and `usb_kbd_open` submits the IRQ URB. When a key is pressed or released, the URB completion callback `usb_kbd_irq` is called. `usb_kbd_irq` parses the data received from the keyboard and reports key-presses and releases to the input API. It then resubmits the URB. When the input API decides the LED status needs to be changed, the `usb_kbd_event` callback is invoked. This callback checks whether a LED URB is in progress, and if not submits the LED URB with the appropriate data. Otherwise, it stores the new LED info in a buffer. When the LED URB completion callback `usb_kbd_led` is called, this callback checks whether new LED info has appeared while the LED URB was in progress. If so, `usb_kbd_led` resubmits the LED URB with the new LED info.

4 Verifying the USB BP Keyboard Driver

Verification of the driver is against the original API. Wrapper functions are only used in a few cases where API functions return a struct (i.e. not a pointer to a struct) because this is currently not supported by VeriFast. The APIs that `usbkbd` uses are the USB API, the input API, spinlocks, and some generic functions like `memcpy`. Verification thus consists of (1) writing formal specifications for these APIs, based on official documentation and reading the API implementation for the underspecified or undocumented parts, and (2) of adding annotations to `usbkbd`. These annotations consists of contracts (pre- and postconditions written in separation logic), predicates to describe data structures, predicate family instances to instantiate callback function contracts, lemmas (i.e. ghost functions), and ghost-code like folding and unfolding predicates.

The verified properties are freedom of data races in the presence of concurrent callbacks, freedom of illegal memory accesses, and correct API usage. This does not include a formal proof of correctness of the hand-written API formalization.

`usbkbd` is one of the smallest Linux kernel drivers. It consists of 426 lines of C code (including blanks and comments). VeriFast reports 329 lines of actual code and 822 lines of annotations. The API specifications count up to 769 lines of code. VeriFast can be launched for this driver with "`verifast -prover redux -c usbkbd_verified.c`". On an Intel L9400 1.86GHz running the verifier takes about one second. The annotated sources of `usbkbd`, specifications for the used APIs and the patches submitted to the driver's maintainer are available at `http://people.cs.kuleuven.be/~willem.penninckx/usbkbd/`.

Writing Specifications for the Input API and some generic functions like `kmalloc` was rather straightforward. API rules include forbidding double frees, requiring when registering input devices that the given callbacks are real function pointers with a contract not conflicting with some rules, etc.

Killable URBs were rather tricky to get verified for the LED URB. Because `usb_kbd_event` and `usb_kbd_led` both submit URBs, they are synchronized

with a spinlock. A C boolean `led_urb_submitted` represents whether the URB is in progress, and thus also whether the URB data (necessary for URB submitting) is not owned by the lock invariant. After killing the URB, the URB data must be taken out of the lock invariant in order to free it, i.e. VeriFast must be convinced `led_urb_submitted` is false. We used a ghost-counter (associated with a predicate of which a uniqueness-proof must be provided on creation) named `cb_out_count` that yields a ticket on increase and ensures the counter is at least n high if n such tickets are owned. Another counter, `killcount`, keeps track of the number of URB submits. By making sure `killcount` tickets of `cb_out_count` are obtained when killing the URB, we can prove `cb_out_count` is at least as high as `killcount`. Because `cb_out_count` is maximum one less than `cb_out_count`, we know they are equal, which can only happen if the URB is not submitted.

The `usb_kbd_malloc` and `usb_kbd_free`'s Contracts take into account all possible combinations of failed and successful allocation and initialization, which makes their contracts long, and dependent on other parts of the annotations.

Flow Between Callbacks had to be reasoned about: permissions are passed between callbacks by setting up callbacks in other callbacks. Reasoning about flow between multiple callbacks easily gives the impression big parts of the program must be taken into account at the same time.

5 Related Work

Here we discuss related case studies and tools in the context of OS verification. The reader is referred to [11] for a discussion of the related work on VeriFast.

Several automated tools for verifying C programs have been introduced. Notably, CEGAR-based [5] model checkers such as BLAST [9] and SLAM/SDV [1] have been applied to check the conformance of device drivers with a set of API usage rules. In contrast with our work, these tools do not provide support for identifying errors with respect to the inherently concurrent execution environment device drivers are operating in. The tools also assume either that a program "does not have wild pointers" [1] or, as shown in [13], perform poorly when checking OS components for memory safety.

In [18] a model checker with support for pointers, bit-vector operations and concurrency is evaluated on a case study on Linux device drivers. The tool checks for buffer overflows, pointer safety, division by zero and user-written assertions. Yet, it requires a test harness with a fixed number of threads to be generated for each driver. VeriFast, in difference, handles concurrency implicitly and aims at verifying full functional correctness and implements assume-guarantee reasoning using generic API contracts. Therefore, VeriFast can check each function of a driver in isolation, which contributes to the scalability of our approach.

Bounded model checking and symbolic execution have been successfully applied to the source code [15, 12] and to the object code [14] of kernel modules. In contrast to the VeriFast approach, these techniques suffer from severe limitations with respect to reasoning about concurrently executing kernel threads.

Shape analysis has been applied to Windows [2] and Linux [19] drivers, and aims to automatically infer, e.g. whether a variable points to a cyclic or acyclic list. Shape analysis can be employed to verify pointer safety, guaranteeing that the shape of data structures is maintained throughout program execution. Ongoing work on VeriFast envisages the use of shape analysis to infer annotations [17].

A competing toolkit to VeriFast is the Verifying C Compiler (VCC) [6]. VCC verifies C programs annotated with contracts in Boogie. The tool generates verification conditions from the annotated program, which are then discharged by an SMT solver. VCC can be expected to require fewer annotations than VeriFast, however, at the expense of a less predictable search times. The toolkit has been employed in a case study on verifying the Microsoft Hypervisor.

Other approaches to OS verification involving modelling and interactive proof. Most notably, the L4.verified [8] project aims at producing a verified OS kernel by establishing refinement relations between several layers of Isabelle/HOL specifications, a prototypic kernel implementation in Haskell and the actual kernel implementation in C and assembly. This differs from our work as we do not employ refinement relations and verification is non-interactive.

6 Conclusions

We report on the successful verification of `usbkbd`, the USB Boot Protocol keyboard driver distributed with the Linux kernel, using the sound and efficient verification tool VeriFast. The verified properties are crash-freedom, race-freedom, and a set of API usage rules. The `usbkbd` driver presents a challenging case study as it involves concurrency and employs a complex API.

VeriFast requires the source code to be annotated with method contracts that are typically easy to write. Certain programming constructs that are difficult to annotate are discussed in this paper. During verification, we identified two bugs related to erroneous synchronization and a missing URB kill. Our case study shows that VeriFast is a powerful tool. Yet, the annotation overhead amounts to a total of 4.8 lines of annotations per line of code. About half of these annotations specify API contracts, that can potentially be reused in future case studies.

Verifying functional correctness and unload-safety is left for further work. Unload-safety includes making sure the kernel does not maintain a function-pointer to a callback of a module that is already unloaded. It is hard to tell whether our verification approach will scale for larger device drivers. More automation for writing or generating annotations with a high degree of decomposition might help. From our experience we conclude that execution speed of the verification tool will not impose problems for larger drivers.

Acknowledgments. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by the Research Fund KU Leuven, and by the EU FP7 projects SecureChange and NESSoS. Jan Smans is a postdoctoral fellow of the Fund for Scientific Research – Flanders (FWO). We acknowledge support from Microsoft Research Cambridge as part of the Verified Software Initiative.

References

1. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, 2006.
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., and Yang, H. Shape analysis for composite data structures. In *CAV 2007*, vol. 4590 of *LNCS*, pp. 178–192, Heidelberg, 2007. Springer.
3. Bornat, R., Calcagno, C., O'Hearn, P., and Parkinson, M. Permission accouting in separation logic. In *POPL*, 2005.
4. Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R. An empirical study of operating system errors. In *SOSP '01*, pp. 73–88, New York, 2001. ACM.
5. Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
6. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. VCC: A practical system for verifying concurrent C. In *TPHOLs '09*, vol. 5674 of *LNCS*, pp. 23–42, Heidelberg, 2009. Springer.
7. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., and Sagiv, M. Local reasoning for storable locks and threads. In *APLAS07*, 2007.
8. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., and Petters, S. M. Towards trustworthy computing systems: taking microkernels to the next level. *SIGOPS Oper. Syst. Rev.*, 41:3–11, July 2007.
9. Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., and Weimer, W. Temporal-safety proofs for systems code. In *CAV '02*, vol. 2402 of *LNCS*, pp. 382–399, Heidelberg, 2002. Springer.
10. Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, 1969.
11. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods 2011*, vol. 6617 of *LNCS*, pp. 41–55, Heidelberg, 2011. Springer.
12. Kim, M. and Kim, Y. Concolic testing of the multi-sector read operation for flash memory file system. In *SBMF '09*, vol. 5902 of *LNCS*, pp. 251–265, Heidelberg, 2009. Springer.
13. Mühlberg, J. T. and Lüttgen, G. BLASTing Linux code. In *FMICS '06*, vol. 4346 of *LNCS*, pp. 211–226, Heidelberg, 2007. Springer.
14. Mühlberg, J. T. and Lüttgen, G. Verifying compiled file system code. In *SBMF '09*, vol. 5902 of *LNCS*, pp. 306–320, Heidelberg, 2009. Springer.
15. Post, H., Sinz, C., and Küchlin, W. Towards automatic software model checking of thousands of Linux modules – a case study with Avinux. *Softw. Test. Verif. Reliab.*, 19:155–172, 2009.
16. Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *LICS '02*, pp. 55–74, Washington, 2002. IEEE.
17. Vogels, F., Jacobs, B., Piessens, F., and Smans, J. Annotation inference for separation logic based verifiers. In *FMOODS 2011*, vol. 6722 of *LNCS*, pp. 319–333, Heidelberg, 2011. Springer.
18. Witkowski, T., Blanc, N., Kroening, D., and Weissenbacher, G. Model checking concurrent Linux device drivers. In *ASE '07*, pp. 501–504, New York, 2007. ACM.
19. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., and O'Hearn, P. Scalable shape analysis for systems code. In *CAV 2008*, vol. 5123 of *LNCS*, pp. 385–398, Heidelberg, 2008. Springer.