Reverse-Engineering Drivers for Safety and Portability

Vitaly Chipounov and George Candea
School of Computer & Communication Sciences
EPFL (Lausanne, Switzerland)

Abstract

Device drivers today lack two important properties: guaranteed safety and cross-platform portability. We present an approach to incrementally achieving these properties in drivers, without requiring any changes in the drivers or operating system kernels. We describe RevEng, a tool for automatically reverse-engineering a binary driver and synthesizing a new, safe and portable driver that mimics the original one. The operating system kernel runs the trusted synthetic driver instead of the original, thus avoiding giving untrusted driver code kernel privileges. Initial results are promising: we reverse-engineered the basic functionality of network drivers in Linux and Windows based solely on their binaries, and we synthesized safe drivers for Linux. We hope RevEng will eventually persuade hardware vendors to provide verifiable formal specifications instead of binary drivers; such specifications can be used to automatically synthesize safe drivers for every desired platform.

1 Introduction

As far as kernel-mode code goes, device drivers are quite buggy. On Linux, for instance, device drivers have $3\times$ to $7\times$ higher bug density than the rest of the kernel code [8]. Most drivers originate either with hardware manufacturers—sole holders of the secrets of a device's internals—or part-time open-source contributors. Writing reliable software and keeping abreast of the latest changes in OS interfaces is not the core competence of a hardware manufacturer, so writing drivers is typically outsourced to third-party companies, which are by now largely commoditized and often do not have a quality reputation to uphold. Not surprisingly, drivers caused 85% of crashes on Windows XP [26] and over one million crashes on Windows Vista [23].

It is ironic then that we are comfortable running such code inside our kernels, especially if we are at all paranoid about viruses, spyware, and other malware. Buggy drivers not only crash systems, but also compromise security. Last year, for example, a zero-day vulnerability was disclosed within a third-party driver shipped with all versions of Windows XP: secdrv.sys, developed by Macrovision as part of SafeDisc [22]. This vulnerability allows non-privileged users to elevate their privileges to Local System, leading to complete system compromise.

Driver safety has garnered a lot of merited attention over the years: microkernels run drivers in user space [14], virtual machine-based approaches isolate drivers from the OS kernel [20, 12, 27, 19], microdrivers reduce the amount of driver code run in the kernel [13], and Nooks can mitigate the consequences of driver failure by isolating user-mode applications from the driver [31]. Many of these approaches are not end-all solutions, but mainly intermediate steps—some require driver source code modifications, others introduce significant performance overheads, etc. More radical approaches aim for drivers that are safe by construction, using domain-specific languages [30, 21, 29]; these too require changes in how drivers are written and do not offer yet a solution for existing drivers.

Besides being unsafe, drivers are also non-portable, because of the close driver/kernel coupling; this hurts both consumers and vendors. Consumers are constrained to one or two OSes if they want good device support, and they are often forced to upgrade to new versions if they want to benefit from new peripherals. Vendors suffer as well, because the cost of porting and supporting drivers on multiple OSes is often prohibitive, so they release drivers only for one or two major platforms, thus restricting the market reach of their products.

Portability, like safety, has also received due attention. Attempts like the Uniform Driver Interface [28] had limited success, mainly because they required close cooperation between hardware vendors. Others, like NDISwrapper [25], were targeted only at specific subsystems.

In this paper we present a new approach to both the safety and portability challenges: RevEng automatically extracts from binary device drivers the protocol for interacting with hardware and then encodes it into a safe driver that can be run in an unmodified kernel. Until hardware vendors themselves start providing open specifications, reverse-engineering can provide a solution.

2 Reverse-Engineering Device Drivers

Reverse-engineering consists of distilling from the binary device driver its essence: the embedded protocol it uses to interact with hardware. This protocol encodes what the driver must do to perform tasks like sending or receiving packets, setting screen resolutions, etc. RevEng proceeds in two phases: First, it records traces of hardware I/O interactions, memory accesses, and executed instructions. Second, it combines the traces with a static analysis of the driver's binary to obtain the protocol state machine. This knowledge is then re-encoded into a safe synthetic driver targeted at the same or different OS. For each class of devices, RevEng relies on a driver template that contains the platform-specific boilerplate for that class; the extracted state machine is then used to "specialize" the boilerplate with the device-specific elements. Templates can be generated with tools like WinDriver [16]. Figure 1 illustrates RevEng's functionality:

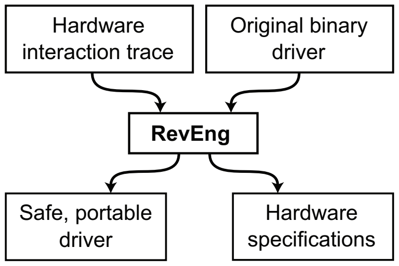

Figure 1: Reverse-engineering drivers with RevEng.

2.1 Extracting the State Machine

Device drivers can be viewed as state machines that encapsulate the protocol for communicating with hardware; RevEng's goal is to extract this state machine. The states of the automaton are snapshots of some of the driver's heap and stack variables. The transition conditions can be predicates on hardware registers or direct kernel invocations of the driver entry points, i.e., the driver functions visible to the OS. Finally, the transition actions result in the driver generating output values that get written to the hardware registers and the kernel.

The driver's internal organization and specifics of the data structures are irrelevant to the protocol state machine. Network driver A may batch incoming packets before delivering them to the OS, while driver B may deliver each one upon arrival; yet, both drivers implement the same driver/hardware protocol, and the hardware cannot distinguish between A and B. Of course, user-perceived performance may differ substantially.

To trace the binary device driver's states and transitions, we use QEMU [3], an open-source virtual machine monitor. The driver runs inside a virtual machine, and RevEng snoops all interactions between the driver and the virtual hardware, traces the program counters of instructions executed by the driver, the register values involved in function calls, and all memory accesses.

RevEng then mines the obtained traces for correlations between inputs provided to the driver and its actions. Consider the following very simple example: a particular register on the network card always has value `0x5b` when a packet is sent, regardless of packet size or destination; this value switches to `0x6b` any time a packet is received. RevEng concludes that sending a packet most likely requires depositing `0x5b` in that particular register and receiving a packet requires value `0x6b`.

We supplement trace analysis with static analysis of the driver's binary. Besides constant writes, drivers may also compute values for the registers, such as a packet length. To reverse-engineer this computation, we find the program slices [33] for those instructions that perform hardware register writes; the slice of such an instruction consists of all the instructions that affect its operands. The program counter traces are used to narrow down the execution path followed by the driver through the slice. RevEng then extracts the logic that computes the value written to the registers. To identify the state (i.e., driver variables) used in the computation, RevEng uses the corresponding memory trace.

RevEng also tracks calls to specific kernel APIs, in order to infer when drivers run asynchronous code via timers, threads, or interrupts. All such operations must be registered with the kernel via specific APIs, to give the kernel the address of the corresponding handler. By recording these calls in the trace, RevEng has sufficient information to identify the asynchronous properties of the original driver and reconstruct them in the synthesized driver. Some asynchronous operations might not have an obvious cause-effect relationship; for example, a driver might decide to switch from asynchronous I/O to polling long after the trigger event occurred. However, by identifying the state that was updated by the trigger event and later used in deciding the switch to polling, RevEng is able to correlate the trigger events with transitions of the driver's state machine.

2.2 Synthesizing New Drivers

To obtain the synthetic executable driver, the slices obtained in the previous step (§2.1) are converted by RevEng into C code, similar to how a decompiler would do it [9]. Memory accesses captured in the trace are replaced with symbolic names in the generated C code. Stack accesses are replaced by local variables. Heap accesses are matched with the traced memory blocks provided by the kernel or allocated by the driver. Instruction and memory traces help resolve pointer aliasing questions and allow memory-mapped I/O to be distinguished from normal memory accesses [10].

The result is a set of C code blocks that represent the reverse-engineered state machine. Executing these RevEng-generated code blocks would result in the same traces as those recorded while snooping on the original driver. The code blocks implement all device-specific actions, thus providing the coupling between the driver and the device. For instance, these code sequences indicate how to send/receive network packets or how to reset the NIC for the network device under study.

The boilerplate that forms a driver template consists of the high-level logic of a driver corresponding to that particular class of devices along with the glue code that couples the driver to the kernel. For instance, a network driver must be able to initialize the network card, send packets, and receive packets. For a given operating system, these operations are invoked via specific kernel functions and follow a specific sequence. All this code would be contained in the network driver template. Currently, all templates are written in C.

Synthesizing a new driver consists of "pasting" the reverse-engineered C code blocks into the driver template, to specialize the boilerplate into a functional driver specific to the device in question. Currently, this specialization is still done manually, but we hope RevEng to eventually do it automatically.

The reverse-engineering process occurs incrementally. A given trace represents one particular execution path through the original driver code, and many basic blocks may not have been exercised. These result in missing blocks of the sate machine; RevEng annotates such blocks with special markers (preprocessor macros) indicating that they correspond to existing functionality that has not been reverse-engineered yet. For example:

```
if (reg2 < 10) {
   pktlen = reg2 + 64 + hdrlen ;
   disable() ;
   outportb( PADR, pktlen ) ;
   enable() ;
} else
   NOT_EXPLORED ;
```

As additional executions cover previously-unexplored basic blocks, the functionality is progressively discovered, and the macros are replaced with reverse-engineered code blocks.

It is also possible to steer the original driver down unexercised paths. For instance, we can compute path constraints using symbolic execution [18] and solve them to obtain input values that will take the driver down the desired paths [4, 17]. RevEng does not support such steering yet. The most difficult paths to exercise are error recovery paths, and we intend to use (virtual) hardware fault injection to reach them. We want to employ both types of steering as part of a feedback loop to dynamically reverse-engineer unexercised paths.

3 Properties of Synthesized Drivers

Five properties are of interest to RevEng: equivalence, completeness, safety, liveness, and portability.

Equivalence: To the hardware, I/O operations performed by the synthesized driver should be indistinguishable from those performed by the original driver. In our current prototype, this generally holds, except we cannot yet reverse-engineer all error recovery paths. So, by generating certain errors, the hardware could tell the two drivers apart. Note that equivalence is not the same as completeness, i.e., the property that the synthesized driver can do everything the original one did.

Completeness: It is not always feasible to completely reverse-engineer a driver. Fortunately, partial reverse-engineering can be quite useful (e.g., having all 2D acceleration in a graphics driver but perhaps not the 3D one). Nevertheless, a future version of RevEng will be able to run the synthesized driver in parallel with the original one, the latter suitably sandboxed in a virtual machine. Requests that cannot be handled by the synthesized driver are relayed to the sandbox; tracing the execution can be used to augment the synthesized driver progressively, until it becomes complete. The state of the two drivers has to be kept synchronized; since state variables have the same layout in both drivers, state can be explicitly copied to the target stack and heap, and execution transferred to the not-yet-reverse-engineered blocks.

This raises the question of when is a synthetic driver ready to replace the original? From a subjective user's perspective, it is when all the desired functionality has been reverse-engineered. Objectively, completeness can be measured as coverage of the original's basic blocks, functions, or code paths. We must also take into account loop and array boundaries, where tracing one iteration or one access may not be enough. For the drivers we reverse-engineered, a naive workload was sufficient to obtain a useful network driver (§4 has more details).

Safety: The synthetic driver arises from merging a driver template with a reverse-engineered state machine. We expect the driver template, whether generated manually or by tools (e.g., WinDriver [16]), to be checked for correctness using formal methods (e.g., with SLAM [2]). This is a worthwhile investment, because templates can be reused across drivers of the same class.

The state machine is assumed safe by construction. RevEng uses recorded traces and any trace that has led to a safety violation (e.g., that resulted in a crash because of a bad pointer) are not used in the reverse-engineering process. As long as all "bad traces" can be excluded as non-safe, the resulting state machine will be safe.

RevEng must be trusted to generate correct code, much the same way a compiler is trusted. Devices with programmable firmware might be sensitive to missing error recovery paths or certain timing characteristics. In general, however, hardware and its drivers are indulgent with respect to timing [34].

Liveness: Infinite loops and deadlocks in drivers would cause the kernel to hang; RevEng ensures that reverse-engineered loops can never become infinite and deadlocks are not encountered. In our instruction traces, loops appear as a sequence of duplicated loop bodies. We found that the Linux driver base and the Windows Driver Kit [24] have only five types of loops: ones with constant number of iterations (typically used to initialize registers), polling loops, delay loops, data transfer loops, and structure traversal loops. RevEng reverse-engineers the first three automatically, although the constant-iterations loops are still kept unrolled. Data transfer loops and structure traversal loops are currently deferred to manual inspection, but we are working on automatically generating these too.

Portability: Driver templates are easy to generate for common classes of devices, because these devices tend to operate in the same manner, e.g., all graphic card drivers set up a framebuffer and perform similar operations on it. Templates are OS-specific. In some cases, it may be worthwhile generating more specialized templates for a particular line of devices from the same manufacturer, enabling quicker support of new models. We have not yet worked with highly specialized custom devices, which might invalidate some of these assumptions.

When a new version of a driver is released, a future version of RevEng will perform a binary diff to identify the added code paths. Suitable workload will then be generated to exercise the modified code paths, similarly to automatic patch-based exploit generation [6].

4 Preliminary Results

We reverse-engineered the Linux NE2000 8390 network device driver and generated a synthetic driver that can reliably initialize the network interface, set the MAC address, and send/receive packets. Performance overhead is negligible both in terms of throughput and latency. For state machine extraction we used a 500 KB trace obtained with a 5-second workload consisting of sending and receiving packets of different sizes. Specializing the template was done manually and took $\sim$4 hours. The stripped binary of the synthetic driver is 12 KB compared to 18 KB for the original driver; the size difference is mostly due to the reduced functionality.

We also used RevEng to port the Windows NE2000 8029AS driver to Linux, using the same workload as above. Manual specialization of the template took considerably longer ($\sim$3 days) and was error-prone, primarily because of the programmer-unfriendly code generated by RevEng and due to the API differences between Linux and Windows kernels. All the integration errors were in the hardware-specific portion of the driver and did not affect the safety of the driver from the point of view of the OS. We are currently working on generating friendlier code and automating the process.

Even low coverage turned out to result in a useful driver. With the 5-second workload we obtained a basic block coverage of 48% and 56% for the Linux and Windows driver, respectively. In Windows, many low-level functions achieved full coverage. However, more complex drivers will likely require higher coverage, if we are to obtain useful synthetic drivers.

5 Discussion

We believe that safe synthetic drivers provide a better way to run privileged code that interacts with hardware: they reduce downtime and security vulnerabilities, and can help kernels promise higher data integrity. Another advantage is portability: imagine "instantly porting" drivers from one platform to another, to the benefit of consumers, who can use all hardware devices with their favorite OS, and the benefit of vendors, who no longer have to invest in providing drivers for multiple platforms. The time and effort savings can be used to build better hardware. All these benefits can be had without changes to any OS kernel. However, there are still a few open questions, which we address next.

5.1 When Is A Driver RevEng-able?

To reverse-engineer a driver, the semantics of its interface with the external world must be sufficiently well understood to connect cause (e.g., the invocation of an entry point) and effect (e.g, a sequence of hardware I/O).

Operations such as `ioctl` can blur this connection. For instance, user-mode applications are often used to configure graphics cards; these applications issue `ioctl`s to the device. A click in the configuration GUI may therefore cause a sequence of `ioctl`s in a way that is entirely user-dependent. We intend to augment RevEng with data flow analysis that will help track the input from the configuration change to the hardware registers, such that we synthesize a driver that preserves the `ioctl`-based interface. This would enable the reuse of the original proprietary user-mode applications.

For some devices (e.g., that are not part of a commonly used class of hardware), producing a template may seem to require more effort than simply writing a driver. Nevertheless, mandatory boilerplate is quite large (e.g., on Windows, power management support must be included for all plug-and-play devices, regardless of whether they need it or not), so separating driver code into template vs. device-specific code is anyway a good idea.

5.2 Extracting Hardware Specifications

In addition to the hardware protocol, RevEng also obtains a specification of the hardware, as encoded in the original device driver. This hardware specification, once translated into a formal language, can be used to verify the assumptions of the original driver implementation against the hardware's actual specification.

To recover the semantics of registers, RevEng records multiple traces while perturbing input parameters (e.g., which mouse button is pressed, the size of data packets, screen resolution). The I/O differences between traces are then correlated to the changing parameter, in a way similar to opcode reverse-engineering [15]. Our current technique requires refinement, though. While effective at recovering register semantics of simple devices, such as a PS/2 mouse, aligning traces to compare registers becomes challenging for complex drivers: the same register might have different meanings depending on context (as in the case of register banks), and the traces can be polluted by unrelated I/O. We are working both on techniques for better filtering of traces and statistical approaches to trace correlation.

5.3 Legal Aspects

Some parts of RevEng resemble decompilation, because we translate original binary code into C. This may have legal implications, if the binary is protected by intellectual property rights, patents, or has an otherwise restrictive license. It could also prevent the use of RevEng as a way to generate synthetic drivers for subsequent redistribution. Employing RevEng for private use, however, should not be problematic; once RevEng is fully automated, private use could be the preferred usage scenario.

Projects connected to reverse-engineering of proprietary software, like Wine [1] or ReactOS [11], have had legal problems in the past. We believe this type of challenges can be mitigated if the extracted code, which we now use to specialize driver templates, is treated as a mere specification of the driver, not as raw code to inject into the template. This would ensure no original code leaks from the original driver to the synthetic one—an approach that conforms to the clean-room principle [5].

5.4 Challenges of Reverse-Engineering

RevEng's current reliance on QEMU introduces certain limitations. First, it is not possible to recover drivers unless an emulation of the corresponding device exists. Second, QEMU does not fully emulate error conditions, like packet transmission errors or seek errors on disk drives; to reproduce such conditions, we would have to interface QEMU with real hardware. Third, QEMU's PCI emulation approach [32] is limited to port I/O and interrupts; however, modern PCI hardware also needs support for DMA and PCI-Express. DMA support can be added using IOMMUs, while PCI-Express support would require the emulation of a virtual chipset interface.

Our current tracing infrastructure introduces a two-fold slowdown in the driver that is being traced, because of the disk accesses for writing the trace; there is no impact on code running outside of the driver, since it is not instrumented, and the synthesized driver's performance is not affected either (§4). However, the tracing slowdown could affect RevEng's ability to reverse-engineer time-sensitive drivers. Acquisition devices, like sound cards, have certain real-time requirements; if tracing is too slow, the driver may end up executing mostly recovery code, due to timeouts. In such cases, we can use hardware tracing solutions [7] instead of virtual machines.

Many of these challenges arise from the fact that we start by reverse-engineering binary device drivers. However, we view reverse-engineering as a stopgap measure, and we hope that a successful RevEng will persuade hardware vendors to provide specifications in a standardized formal hardware abstraction language (HAL) instead of binary drivers. Some of them already provide informal specifications in datasheets. Others may need to reconsider how they develop their hardware interfaces, to prevent competitors from inferring sensitive information about their chips. Once HAL specifications are available, RevEng can generate safe, verified drivers for any platform of interest.

6 Conclusion

We proposed a new approach to solving the problem of safety and portability of device drivers, without requiring access to source code or any modifications to hardware, drivers, or existing operating systems. RevEng takes the original binary driver, executes it in a virtual machine, and traces its interaction with hardware. The traces are then used to extract the hardware interaction protocol embodied in the driver. While RevEng can provide safety and portability today, our hope is that eventually hardware vendors will migrate to a model in which they release formal specifications of the hardware interaction protocols, instead of closed binary drivers. Once they do, the world will be a better place: operating systems will crash less, new devices will be supported on all platforms, and the OS playing field will become more level. Hardware obsolescence will slow down and there will be fewer forced upgrades and unjustified costs.

7 Acknowledgments

We would like to thank Willy Zwaenepoel, Mike Swift, Aravind Menon, the members of DSLab, as well as the anonymous reviewers for their feedback on this work.

References

[1] B. Amstadt, E. Youngdale, and A. Julliard. Wine is not an emulator. http://www.winehq.org/.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, 2006.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX*, 2005.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *ISSTA*, 2002.

[5] D. D. F. Brewer and D. M. J. Nash. The Chinese wall security policy. In *SP*, 1989.

[6] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *SP*, 2008.

[7] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. 2006.

[8] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. 2001.

[9] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, School of Computing Science, 1994.

[10] C. Cifuentes, T. Waddington, and M. V. Emmerik. Computer security analysis through decompilation and high-level debugging. In *Working Conf. on Reverse Engineering*, 2001.

[11] S. Edwards. Reset, reboot, restart, legal issues and the long road to 0.3. http://www.reactos.org/en/news_page_14.html, 2006.

[12] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williams. Safe hardware access with the Xen virtual machine monitor. In *WOSAS*, 2004.

[13] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *ASPLOS*, 2008.

[14] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of µ-kernel-based systems. In *SOSP*, 1997.

[15] W. C. Hsieh, D. R. Engler, and G. Back. Reverse-engineering instruction encodings. In *Proc. USENIX Annual Technical Conference*, 2001.

[16] Jungo. WinDriver device driver development tookit, version 9.0. http://www.jungo.com/windriver.html, 2007.

[17] N. Kicillof, W. Grieskamp, N. Tillmann, and V. Braberman. Achieving both model and code coverage with automated gray-box testing. In *Proc. 3rd International Workshop on Advances in Model-based Testing*, 2007.

[18] J. C. King. Symbolic execution and program testing. *CACM*, 1976.

[19] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical report, Universität Karlsruhe (TH), 2005.

[20] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI*, 2004.

[21] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, , and G. Muller. Devil: An IDL for hardware programming. In *OSDI*, 2000.

[22] Microsoft security advisory #944653: Vulnerability in macrovision driver. http://www.microsoft.com/technet/security/advisory/944653.mspx.

[23] Microsoft. Microsoft internal memo, provided as public evidence in court case #c07-475mjp. http://blog.seattlepi.nwsource.com/microsoft/library/ vistaexhibitsone.pdf, 2008.

[24] Microsoft Windows Driver Kit. http://www.microsoft.com/whdc/devtools/WDK, 2009.

[25] NDISwrapper. http://ndiswrapper.sourceforge.net, 2008.

[26] V. Orgovan and M. Tricker. An introduction to driver quality. Microsoft Windows Hardware Engineering Conf., 2003.

[27] B. Poess. Binary device driver reuse. Master's thesis, Universität Karlsruhe (TH), 2007.

[28] Project UDI. Uniform Driver Interface. http://udi.certek.com/, 2008.

[29] M. F. Spear, T. Roeder, O. Hodson, G. C. Hunt, and S. Levi. Solving the starting problem: device drivers as self-describing artifacts. In *EuroSys*, 2006.

[30] J. Sun, W. Yuan, M. Kallahalla, and N. Islam. HAIL: a language for easy and correct device access. In *EMSOFT*, 2005.

[31] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, 24(4):333–360, 2006.

[32] G. Tedesco. QEMU Host PCI Proxy V0.3 patch. http://ml.osdir.com/emulators.qemu/2004-06/msg00365.html.

[33] M. Weiser. Program slicing. In *ICSE*, 1981.

[34] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *OSDI*, 2008.