Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability

Reinhard Tartler[1], Anil Kurmus[2],
Bernhard Heinloth[1], Valentin Rothberg[1], Andreas Ruprecht[1], Daniela Dorneanu[2],
Rüdiger Kapitza[3], Wolfgang Schröder-Preikschat[1], and Daniel Lohmann[1]

[1]*Friedrich-Alexander University Erlangen-Nürnberg*
[2]*IBM Research - Zurich*
[3]*TU Braunschweig*

Abstract

The Linux kernel can be a threat to the dependability of systems because of its sheer size. A simple approach to produce smaller kernels is to manually configure the Linux kernel. However, the more than 11,000 configuration options available in recent Linux versions render this a demanding task. We report on designing and implementing the first automated generation of a workload-tailored kernel configuration and discuss the security gains such an approach offers in terms of reduction of the Trusted Computing Base (TCB) size. Our results show that the approach prevents the inclusion of 10% of functions known to be vulnerable in the past.

1 Introduction

The Linux kernel is a commonly attacked target. In 2011, 148 Common Vulnerabilities and Exposures (CVE)[1] entries for Linux have been recoded, and this number is expected to grow every year. This is a serious problem for system administrators who rely on a distribution-maintained kernel for the daily operation of their systems. On the Linux distributor side, kernel maintainers can make only very few assumptions on the kernel configuration for their users: Without a specific use case, the only option is to enable every available configuration option to maximize the functionality. The ever-growing kernel code size, caused by the addition of new features, such as drivers, file systems and so on, indicates that the risk of undetected vulnerabilities will constantly increase in the foreseeable future.

If the intended use of a system is known at kernel compilation time, an effective approach to reduce the kernel's attack surface is to configure the kernel to not compile unneeded functionality. However, finding a fitting configuration requires extensive technical expertise about currently more than 10,000 Linux configuration options, and needs to be repeated at each kernel update. Therefore, maintaining such a custom-configured kernel entails considerable maintenance and engineering costs.

This paper presents a tool-assisted approach to automatically determine a kernel configuration that enables only kernel functionalities that are actually necessary in a given scenario. We quantify the security gains in terms of reduction of the Trusted Computing Base (TCB) size. The evaluation section (Section 3) focuses on an appliance-like virtual machine that runs a web server similar to those used to power large distributed web services in the cloud. Our approach exhibits promising security improvements for this use case: Compared with a default distribution kernel, 10% of the kernel functions (i.e., 17 out of 179), for which in total 31 vulnerabilities have been reported, are removed from the tailored kernel.

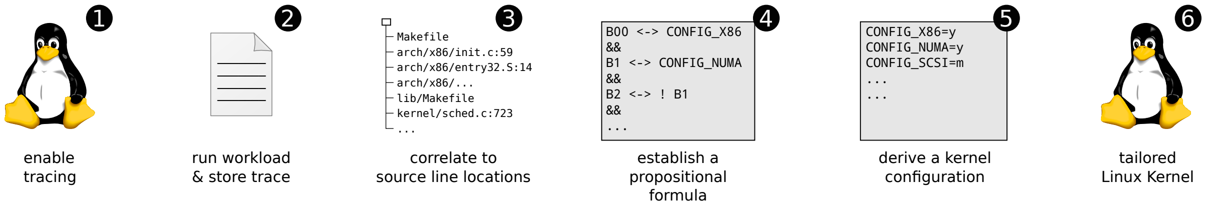
The remainder of this paper is structured as follows: Section 2 presents the design and implementation of the first automated workload-specific kernel-build generation tool. Section 3 evaluates of the usability of such an approach in a real-world scenario. Security benefits of the tailored Linux kernel are discussed in Section 4. Section 5 presents the related work. The paper concludes in Section 6.

2 Kernel-Configuration Tailoring

The goal of our approach is to compile a Linux kernel with a configuration that has only those features enabled which are necessary for a given use case. This section shows the fundamental steps of our approach to tailor such a kernel. The six necessary steps are depicted in Figure 1.

➊ **Enable tracing.** The first step is to prepare the kernel so that it records which parts of the kernel code are executed at run time. We use the Linux-provided `ftrace` feature, which is enabled with the KCONFIG configuration option `CONFIG_FTRACE`. Enabling this configuration option modifies the Linux build process to include profiling

[1] `http://cve.mitre.org/`


❶
enable tracing
❷
run workload & store trace
❸
correlate to source line locations
❹
establish a propositional formula
❺
derive a kernel configuration
❻
tailored Linux Kernel

```
Makefile
arch/x86/init.c:59
arch/x86/entry32.S:14
arch/x86/...
lib/Makefile
kernel/sched.c:723
...
```

```
B00 <-> CONFIG_X86
&&
B1 <-> CONFIG_NUMA
&&
B2 <-> ! B1
&&
...
```

```
CONFIG_X86=y
CONFIG_NUMA=y
CONFIG_SCSI=m
...
...
```

Figure 1: Workflow of the approach

code that can be evaluated at runtime.

In addition, our approach requires a kernel built with debugging information so that any function addresses in the code segment can be correlated to functions and thus source file locations in the source code. For Linux, this is configured with the Kconfig configuration option `CONFIG_DEBUG_INFO`.

❷ **Run workload.** In this step, the system administrator runs the targeted application after enabling `ftrace`. The `ftrace` feature now records all addresses in the text segment that have been instrumented. For Linux, this covers most code, except for a small amount of critical code such as interrupt handling, context switches and the tracing feature itself.

To avoid overloading the system with often accessed kernel functions, `ftrace`'s own ignore list is dynamically being filled with functions when they are used. This prevents such functions from appearing more than once in the output file of `ftrace`. We use a small wrapper script for `ftrace` to set the correct configuration before starting the trace, as well as to add functions to the ignore list while tracing and to parse the output file, printing only addresses that have not yet been encountered.

❸ **Correlation to source lines.** A system service translates the raw address offsets to source line locations using the addr2line tool from the `binutils` tool suite. This identifies the source files and the `#ifdef` blocks that are actually being executed during the tracing phase. Technically, the tool stores its result to a text file with source-file names and line numbers on each line.

❹ **Establishment of the propositional formula.** This step translates the source-file locations into a propositional formula. The propositional variables of this formula are the *variation points* the Linux configuration tool Kconfig controls during the compilation process. This means that every C Preprocessor (CPP) block, Kconfig item and source file can appear as propositional variable in the resulting formula. This formula is constructed with the variability constraints that have been extracted from `#ifdef` blocks, Kconfig feature descriptions and Linux Makefiles. The extractors we use have been developed, described and evaluated in previous work [5, 19, 20]. The resulting formula holds for every Kconfig configuration that enables all source lines simultaneously.

❺ **Derivation of a tailored kernel configuration.** A SAT checker proves the satisfiability of this formula and returns a concrete configuration that fulfills all these constraints as example. Note that finding an optimal solution to this problem is an NP-hard problem and was not the focus of our work. Instead, we rely on heuristics and configurable search strategies in the SAT checker to obtain a sufficiently small configuration.

As the resulting kernel configuration will contain some additional unwanted code, such as the tracing functionality itself, the formula allows the user to specify additional constraints to force the selection (or deselection) of certain Kconfig features, which can be specified in whitelists and blacklists. This results in additional constraints being conjugated to the formula just before invoking the SAT checker.

❻ **Compiling the kernel.** The resulting solution to the propositional formula, obtained as described above, can only cover Kconfig features of code that has been traced. As the Kconfig feature descriptions declare non-trivial dependency constraints [23], special care must be taken to ensure that as many Kconfig features as possible are not selected while still fulfilling all dependency constraints. We therefore use the Kconfig tool itself to process this feature selection to a Kconfig configuration that is both consistent and selects as few features as possible.

3 Practical Application

We evaluate the usefulness of our approach by setting up a Linux, Apache, Mysql and PHP (LAMP)-based web presence in a manner that is suited for deployment in a cloud environment. The system serves static webpages, the collaboration platform DokuWiki [6] and the message board system phpBB3 [17] as an example for typical real-world applications. We use the distribution-provided packages from the Debian distribution without further specific configuration changes or optimization. Evaluation results are summarized in Table 1.

3.1 Kernel Tailoring

To derive a minimized kernel configuration, the first step consists of compiling a tracing-enabled Linux kernel. We use the standard Linux kernel source and configuration from the Debian distribution (version `2.6.32-41squeeze2`) as a template for our tracing kernel (Step ❶ in Figure 1). On this kernel, we enable the features `CONFIG_FTRACE` and `CONFIG_DEBUG_INFO` to include the `ftrace` tracing infrastructure and compile with debugging symbols. As our current prototype is not able to resolve functions from loadable kernel modules (LKMs) yet, we disable module support in the kernel configuration, which causes all compiled code to be loaded into the system at boot time.

Furthermore, a number of drivers cause compilation and linking errors when not compiled as LKMs. Most of these issues stem from drivers in the `staging`[2] area. Also, when trying to boot this kernel, we observe kernel panics during the initialization of a range of watchdog drivers. As these drivers turn out to be unnecessary for this application scenario, we turn off the KCONFIG options `CONFIG_STAGING` and `CONFIG_WATCHDOG`. These configuration changes account for the difference in size and features between the kernel shipped with Debian (∼42 MB of code in the `text` segment) and the intermediary kernel that is used for collecting traces (∼36 MB of code in the `text` segment).

With this intermediary tracing kernel, the system is tested against a test workload that covers all required functionality. We use the Skipfish [22] security analysis tool to systematically access all functionality of the appliance in an automated manner. This corresponds to Step ❷ in Figure 1 and results in a total of 5,377 observed kernel functions.

These traced kernel functions correlate to 4,686 different source lines in 379 source files (Step ❸). We use a modified version of the UNDERTAKER tool [20] to establish the propositional formula (Step ❹) and to derive a solution for it (Step ❺). To avoid unwanted functionality enabled in the resulting kernel, such as the `ftrace` infrastructure itself and LKM support, the UNDERTAKER tool obeys a blacklist that consists of the KCONFIG options `CONFIG_FTRACE` and `CONFIG_MODULES`. Also, we add eight additional,[3] use-case–agnostic KCONFIG items to the whitelist in Step ❺ to enable features that are used by the initialization startup scripts, which run before the system-wide tracing process starts. These steps take 69 sec on a commodity 2.8 GHz quad-core workstation with 4 GB of RAM.

Kernel Shipped by Debian	
Loaded Code	5,465,602 Bytes
Total Loadable Code	42,188,538 Bytes
Loaded Kernel Modules	29
KCONFIG options set to `y`	1,093
KCONFIG options set to `m`	2,299
Functions with CVE entries	179
Intermediary kernel used for tracing	
Loaded Code	36,341,888 Bytes
Total Loadable Code	36,341,888 Bytes
Loaded Kernel Modules	0
KCONFIG options set to `y`	3,298
KCONFIG options set to `m`	0
Functions with CVE entries	207
Resulting application-tailored kernel	
Loaded Code	3,990,153 Bytes
Total Loadable Code	3,990,153 Bytes
Loaded Kernel Modules	0
KCONFIG options set to `y`	379
KCONFIG options set to `m`	0
Functions with CVE entries	162

Table 1: Results of the experiment at a glance. The code sizes were obtained with the SIZE tool from the BINUTILS suite by adding the sizes of the text segments of the bootable kernel image and all loadable `.ko` files.

3.2 Evaluation

To ensure the functionality of the appliance, we run the Skipfish [22] security scan again on the system with the tailored kernel, and compare the results with the previous run on the tracing kernel. The comparison of these two reports indicates no differences in the number of vulnerabilities or other issues.

The performance is tested with the httperf tool [16]. The tool accesses a static website continuously, at a constant number of requests per second in each run. We did two setups of the same test scenario, both times using the same system, but once booted with the Debian standard kernel, and once with our tailored kernel. The data shows that our tailored kernel achieves a performance very similar to that of the original kernel provided by the distribution.

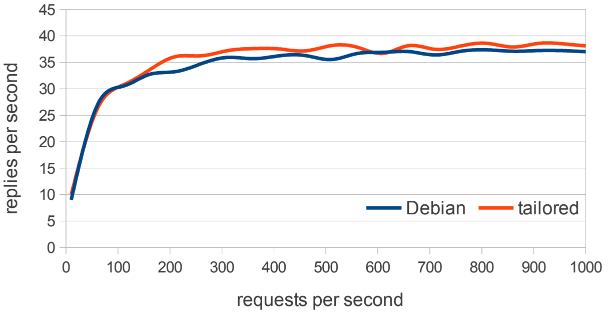


Figure 2: Comparison of reply rates of the web server with the tailored kernel and the standard distribution kernel.

[2] The `staging` area contains unfinished and incomplete drivers that are included as a technology preview.

[3] Specifically: `CONFIG_ACPI`, `CONFIG_UNIX`, `CONFIG_DEVTMPFS`, `CONFIG_DEVTMPFS_MOUNT`, `CONFIG_SERIAL_8250_CONSOLE` and `CONFIG_INOTIFY_USER`, `CONFIG_PM`

4 Discussion

After the presentation of a practical use case for our approach, this section now evaluates the security benefits. For this, we present an applicable security model to determine the TCB, and discuss security improvements in terms of TCB reduction.

Security Model. In the context of the web service presented, we assume both local and remote malicious attackers that target the kernel. However, we do not consider attackers that have physical access to the machine nor attacks that directly target hardware and firmware vulnerabilities.

The security goal is to prevent an attacker from gaining full control with arbitrary code execution in kernel mode, information leakage (e.g., recover uninitialized kernel memory content) to breach confidentiality, and denial-of-service attacks by crashing the kernel to reduce the availability of the system.

TCB sizes. Following the literature [11], we define the TCB as “the subset of components that need to be trusted to fulfill the security goals given in the security model”. Therefore, in the security model above, the TCB is solely composed of the kernel, including all LKM loaded during normal operation.

We apply three different metrics to measure the TCB reduction: a) the compiled code (text segment) size of the kernel, b) the total number of features that are enabled in Kconfig and c) the number of functions compiled into the kernel for which there has been an CVE entry in the past 10 years. More precisely, through a semi-automated process, we map a subset of 197 out of 873 CVE entries to vulnerabilities in 215 unique functions in the kernel, and use this dataset. The results for all three kernels used in the experiment in Section 3 are shown in Table 1.

Results. The data shows that the Linux kernel shipped by Debian loads 5.5 MB of program code into the memory for the virtual machine in the scenario described in Section 3. Compared with the code size of 4 MB for our tailored kernel, the total TCB size is reduced by 27%.

The number of features enabled is also reduced significantly, from 3,392 (with 1,093 features compiled statically into the kernel and 2,299 as LKM) to 379. The omission of functionality to load further LKMs constitutes an additional security benefit.

Finally, for each function in the TCB, we record the number of known vulnerabilities that have been reported in the past 10 years. When comparing the default distribution kernel to the tailored kernel, we observe a reduction of 10% of functions for which vulnerabilities have been reported in the past. However, this number is a lower-bound estimate, as the Linux kernel supports on-demand insertion of LKM, resulting in a higher initial TCB size, and therefore higher TCB reduction.

Sampling bias. Compared with the code size reduction results above, the CVE reduction numbers may seem lower than expected. We hypothesize that this impression can be attributed to sampling bias: code that is used more often is also audited more often, and better care is taken in documenting the vulnerabilities of such functions. A comparison of the average number of CVEs in kernel functions that are loaded and used (9.8‰) with the average number of CVEs in kernel functions that are not used (3.7‰) supports this hypothesis. Previous studies [3] have also shown that code in the `driver/` sub-directory of the kernel, which is known to contain a significant number of rarely-used code, on average contains significantly more bugs than any other parts of the kernel tree. Consequently, it is likely that unused features provided by the kernel still contain a significant amount of relatively easy-to-find vulnerabilities. This further confirms the importance of reducing the TCB size as presented in this work.

Unexpected impacts. The tailoring approach presented in this work as a security solution could in turn cause a reduction of the security of the system – a drawback that is common to many security software but is often overlooked. Reviewing the process described in Section 2 (Step ❻), we cannot rule out that for some application scenarios, performance-critical or security features might be removed from the base kernel. Possible reasons for this include that a) the feature was not triggered during the system-wide trace, b) the functionality has been excluded from the instrumentation with `ftrace` (e.g., for performance reasons), or c) the configuration options influence the resulting kernel in non-functional ways (e.g., different compilation flags, etc.). Although we were not able to find any results confirming this in this experiment — for example, we have verified that the `CONFIG_CC_STACKPROTECTOR` configuration option, which toggles the inclusion of the GCC flag for adding a stack frame canary, remains enabled, in future work we intend to further evaluate potential adverse impacts.

Applicability. The approach presented relies on the assumption that the use-case of the system is clearly defined. Thanks to this a priori knowledge, it is possible to determine what kernel functionalities the application requires and therefore, what kernel configuration options have to be enabled. With the increasing importance of compute clouds, where customers employ virtual machines for very dedicated services such as the web server presented in Section 3, we expect that our approach can be easily applied to further use cases that are commonly deployed in the cloud.

Usability. Most of the steps presented in Section 2 require no domain specific knowledge of Linux internals. We therefore expect that they can be conducted in a straightforward manner by system administrators without specific experience in Linux kernel development. The system administrator, however, continues to use a code base that continuously receives maintenance in form of bug fixes and security updates from the Linux distributor. We therefore are confident that our approach to automatically tailor a kernel configuration for specific use-cases is both practical and feasible to implement in real-world scenarios.

Extensibility. The experiment in Section 3 shows that the resulting kernel requires eight additional KCONFIG options for proper operation. Alternatively to adding these features to the whitelist with distribution-specific knowledge, starting the application tracer at the start of the boot process would also capture the missing functionality. However, in this way we demonstrate the ability to specify wanted or unwanted KCONFIG options independently of the tracing. This allows our approach to be assisted in the future by methods to determine kernel features that tracers such as `ftrace` cannot observe at all.

5 Related work

As we show below, this work relates to two research areas.

Kernel specialization. Several researchers have suggested approaches to tailor the Linux kernel, although security is usually not a goal, but improvements in code size or execution speed are: Lee et al. [12] manually modify the source code (e.g., by removing unnecessary system calls) based on a static analysis of the applications and the kernel. Chanet et al. [2], in contrast, propose a method based on link-time binary rewriting, but also employ static analysis techniques to infer and specialize the set of system calls to be used. Both approaches, however, do not leverage any of the built-in configurability of Linux to reduce unneeded code. Moreover, our approach is completely automated.

TCB reduction has always been a major design goal for micro-kernels [1, 13], which in turn facilitates a formal verification of the kernel [9] or its implementation in typelafe languages, such as OCaml [14].

Kernel attack surface reduction. The security model used in this paper is commonly used when building *sandboxing* or *isolation* solutions, in which each process must be contained within a particular security domain, such as [4, 8, 15], which are all based on the Linux Security Module (LSM) framework [21]. The idea of directly restricting the system call interface on a per-process basis was first presented by Provos [18] for OpenBSD, although not with specific focus on reducing the kernel's attack surface. Seccomp [7] directly tackles this issue by allowing processes to be sandboxed at the system call interface. Ktrim [10] is a current research project which goes beyond simply limiting the system call interface, and explores the possibility of finer-granularity kernel attack surface reduction by restricting individual functions (or sets of functions) inside the kernel. In contrast, this work focuses on compile-time removal of functionality from the kernel at a system-wide level instead of a runtime removal at a per-application level.

6 Conclusion and Future Work

This paper presents an approach for automatically tailoring a Linux kernel configuration to a given use case. The result is a Linux kernel in which unnecessary functionality is removed at compile-time, hence significantly reducing TCB size. The reduction can be quantified with 27% less code loaded and at least 10% fewer kernel functions which were previously vulnerable to attacks.

While the current prototype shows promising results, we intend to improve on the usability and applicability to additional use-cases. For instance, the current prototype unconditionally disables module loading support. As this may be undesirable in some cases, we intend to improve the handling of LKMs, as well as to remove the the need for an intermediary tracing kernel.

Acknowledgments

This research has been partially supported by the TClouds project[4] funded by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement number ICT-257243 and by the German Research Council (DFG) under grant no SCHR 603/7-2 and LO 1719/2-2.

[4] `http://www.tclouds-project.eu`

References

[1] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. “MACH: A New Kernel Foundation for UNIX Development”. In: *Proceedings of the USENIX Summer Conference*. USENIX Association, 1986, pages 93–113.

[2] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. “System-wide Compaction and Specialization of the Linux Kernel”. In: *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES ’05)*. ACM Press, 2005, pages 95–104. ISBN: 1-59593-018-3. DOI: `10.1145/1065910.1065925`.

[3] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. “An empirical study of operating systems errors”. In: *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP ’01)*. (Banff, Alberta, Canada). Edited by Keith Marzullo and M. Satyanarayanan. ACM Press, 2001, pages 73–88. ISBN: 1-58113-389-8. DOI: `10.1145/502034.502042`.

[4] Kees Cook. *Yama LSM*. 2010. URL: `http://lwn.net/Articles/393012/` (visited on 06/04/2012).

[5] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “A Robust Approach for Variability Extraction from the Linux Build System”. In: *Proceedings of the 16th Software Product Line Conference (SPLC ’12)*. (Salvador, Brazil, Sept. 2–7, 2012). (To appear). ACM Press, 2012.

[6] Andreas Gohr. *DokuWiki*. URL: `http://dokuwiki.org` (visited on 06/03/2012).

[7] *Google Seccomp Sandbox for Linux*. URL: `http://code.google.com/p/seccompsandbox/wiki/overview` (visited on 06/05/2012).

[8] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. “Task Oriented Management Obviates Your Onus on Linux”. In: *Proceedings of the Japan Linux Conference* (2004). ISSN: 1348-7868.

[9] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. “seL4: formal verification of an OS kernel”. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP ’09)*. (Big Sky, Montana, USA). ACM Press, 2009, pages 207–220. ISBN: 978-1-60558-752-3. DOI: `10.1145/1629575.1629596`.

[10] Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. “Attack surface reduction for commodity OS kernels: trimmed garden plants may attract less bugs”. In: *Proceedings of the 4th European Workshop on system security (EUROSEC ’11)*. (Salzburg, Austria). ACM Press, 2011, 6:1–6:6. ISBN: 978-1-4503-0613-3. DOI: `10.1145/1972551.1972557`.

[11] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. “Authentication in distributed systems: theory and practice”. In: *ACM Transactions on Computer Systems* 10.4 (1992), pages 265–310. ISSN: 0734-2071. DOI: `10.1145/138873.138874`.

[12] C.T. Lee, J.M. Lin, Z.W. Hong, and W.T. Lee. “An Application-Oriented Linux Kernel Customization for Embedded Systems”. In: *Journal of information science and engineering* 20.6 (2004), pages 1093–1108. ISSN: 1016-2364.

[13] Jochen Liedtke. “On μ-Kernel Construction”. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP ’95)*. ACM SIGOPS Operating Systems Review. ACM Press, 1995. DOI: `10.1145/224057.224075`.

[14] A. Madhavapeddy, R. Mortier, R. Sohan, T. Gazagnaire, S. Hand, T. Deegan, D. McAuley, and J. Crowcroft. “Turning Down the LAMP: Software Specialisation for the Cloud”. In: *Proceedings of the 2nd USENIX Conference on hot topics in cloud computing (HOTCLOUD’10)*. USENIX Association, 2010, pages 11–11.

[15] Frank Mayer, Karl MacMillan, and David Caplan. *SELinux By Example: Using Security Enhanced Linux*. Prentice Hall PTR, 2006, page 456. ISBN: 978-0131963696.

[16] David Mosberger and Tai Jin. “httperf. A tool for measuring web server performance”. In: *SIGMETRICS Performance Evaluation Review* 26.3 (1998), pages 31–37. ISSN: 0163-5999. DOI: `10.1145/306225.306235`.

[17] *phpBB. Free and Open Source Forum Software*. URL: `www.phpbb.com` (visited on 06/03/2012).

[18] Niels Provos. “Improving host security with system call policies”. In: *Proceedings of the 12th Conference on USENIX Security Symposium (SSYM ’03)*. Volume 12. USENIX Association, 2003, pages 18–18.

[19] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. “Efficient Extraction and Analysis of Preprocessor-Based Variability”. In: *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE ’10)*. (Eindhoven, The Netherlands). Edited by Eelco Visser and Jaakko Järvi. ACM Press, 2010, pages 33–42. ISBN: 978-1-4503-0154-1. DOI: `10.1145/1868294.1868300`.

[20] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. “Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem”. In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys ’11)*. (Salzburg, Austria). Edited by Christoph M. Kirsch and Gernot Heiser. ACM Press, 2011, pages 47–60. ISBN: 978-1-4503-0634-8. DOI: `10.1145/1966445.1966451`.

[21] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. “Linux Security Module Framework”. In: *Proceedings of the Ottawa Linux Symposium*. (Ottawa, OT, Canada). Edited by Andrew J. Hutton, Stephanie Donovan, and C. Craig Ross. 2002, pages 604–617.

[22] Michal Zalewski, Niels Heinen, and Sebastian Roschke. *skipfish. Web application security scanner*. URL: `http://code.google.com/p/skipfish/` (visited on 06/03/2012).

[23] Christoph Zengler and Wolfgang Küchlin. “Encoding the Linux Kernel Configuration in Propositional Logic”. In: *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration 2010*. Edited by Lothar Hotz and Alois Haselböck. 2010, pages 51–56.