# Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring

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

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

## Abstract

*The* economy of mechanism *security principle states that program design should be kept as small and simple as possible. In practice, this principle is often disregarded to maximize user satisfaction, resulting in systems supporting a vast number of features by default, which in turn offers attackers a large code base to exploit. The Linux kernel exemplifies this problem: distributors include a large number of features, such as support for exotic filesystems and socket types, and attackers often take advantage of those.*

*A simple approach to produce a smaller kernel is to manually configure a* tailored *Linux kernel. However, the more than* 11,000 *configuration options available in recent Linux versions make this a time-consuming and non-trivial task. We design and implement an automated approach to produce a kernel configuration that is adapted to a particular workload and hardware, and present an attack surface evaluation framework for evaluating security improvements for the different kernels obtained. Our results show that, for real-world server use cases, the attack surface reduction obtained by tailoring the kernel ranges from about 50% to 85%. Therefore, kernel tailoring is an attractive approach to improve the security of the Linux kernel in practice.*

## 1 Introduction

The Linux kernel is a commonly attacked target. In 2011 alone, 148 Common Vulnerabilities and Exposures (CVE)[1] entries for Linux have been reported, 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 and file systems, at an increasing pace, indicates that the Linux kernel will be subject to ever more vulnerabilities.

In addition, as a consequence of the development, testing, and patching process of large software projects, the less a functionality is used, the more likely it is to contain defects. Indeed, developers mostly focus on fixing issues that are reported by their user base. As rarely used functionalities only account for reliability issues in a small portion of the user base, this process greatly improves the overall reliability of the software. However, malicious attackers can, and do, still target vulnerabilities in those less-often-used functionalities. A recent example from the Linux kernel is an arbitrary kernel memory read and write vulnerability in the reliable datagram sockets (RDS) (CVE-2010-3904), a rarely used socket type.

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 include unneeded functionality. However, finding a suitable configuration requires extensive technical expertise about currently more than 11,000 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.

Moreover, while it is widely accepted that making programs "smaller" improves security, quantitatively measur-

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

ing security improvements remains a difficult and important problem [49]. Existing work on system security often measures improvements in terms of Trusted Computing Base (TCB) reduction, which in practice often translates into a measurement of the total number of source lines of code (SLOC) (e.g., [19, 34]). Although these metrics are sensible (as every line of code can have a vulnerability) and easy to obtain, they can be imprecise. For instance, on a given kernel configuration, a large part of the kernel sources will not be compiled, many parts will only be compiled as kernel modules which might never be loaded, and some functions might simply not be within reach of an attacker.

This paper presents metrics for quantifying the security of an OS kernel and a tool-assisted approach to automatically determine a kernel configuration that enables only kernel functionalities that are actually necessary in a given scenario. Although it is easy to quantify the size of the resulting kernel binaries, this is not convincing evidence that the resulting kernel indeed presents less of an attack surface to potential attackers. Hence, after defining what attack surface means, we quantify the security gains in two distinct security models in terms of attack surface reduction. The first security model considers that the entire kernel can be subject to attacks and is therefore a good reference for comparison to previous work, whereas the second considers the scenario of a restricted attacker, and is a good reference for evaluating the security improvements of configuration tailoring in the context of unprivileged local attackers. Our measurements take into account the static call graph of the kernel and the possible entry points of the attacker to provide a more accurate comparison.

Our automated kernel-tailoring approach builds on our previous work [51], and extends it with multiple improvements, including loadable kernel module (LKM) support. When compared to other hardening solutions, a notable advantage of the kernel-configuration tailoring approach is that it makes no changes to the source code of the kernel: therefore, it is impossible to introduce new defects into the kernel source. This approach uses run-time traces as input for deducing a suitable kernel configuration, and we show it to work equally well in different use cases. We detail the use our tool to tailor a "Linux, Apache, MySQL and PHP (LAMP) stack" kernel on server hardware, as well as a network file system (NFS) running on a workstation. We obtain comparative measurements of the tailored kernels that show that configuration-tailoring incurs no overhead and no stability issues, while greatly reducing the attack surface in both security models.

The major contributions of this paper are:

- A definition of an attack surface and an attack-surface metric based on static call graphs and security models; examples of metrics satisfying this definition, and a comparison of the effects of these choices on our measurements.
- A tool that, given the kernel sources and run-time traces characterizing a use case, produces a small kernel configuration, taking into account LKMs, which includes all kernel functionalities necessary for the workload.
- An evaluation of the attack surface reduction as well as performance results in the case of a LAMP-based server and a workstation providing access to files via NFS.

The remainder of this paper is structured as follows: Section 2 defines the notions of attack surface and attack surface measurement, as well as a set of attack surface metrics that can be used in practice for our evaluation. Section 3 presents an overview of the tailoring process, and the implementation of the underlying automated kernel-configuration-tailoring tool. Section 4 evaluates the attack surface reduction and performance of such an approach in two use cases and with several attack surface reduction metrics. These results are then discussed in Section 5. Section 6 presents related work. The paper concludes in Section 7.

## 2 Security Metrics

In this section, we present two distinct security models, and, for each of them, security metrics (attack surface measurements) which we use in Section 4 to evaluate and quantify the security of a running Linux kernel. The dependence between notions defined or used in this section are summarized in Figure 1.

### 2.1 Preliminary definitions

**Definition 1** (Call graph)**.** A *call graph* is a directed graph $(F,C)$, where $F \subseteq \mathbb{N}$ is the set of nodes and represents the set of functions as declared in the source of a program, and $C \subseteq F \times F$ the set of arcs, which represent all direct and indirect function calls. We denote the set of all call graphs by $\mathscr{G}$.

In practice, static source code analysis at compile time (that takes all compile-time configuration options into account) is used to obtain such a call graph.

**Definition 2** (Entry and barrier functions)**.** A security model defines a set of *entry functions* $E \subseteq F$, which corresponds to the set of functions directly callable by an attacker, and a set of *barrier functions* $X \subseteq F$, which corresponds to the set of functions that, even if reachable, would prevent an attacker from progressing further into the call graph.

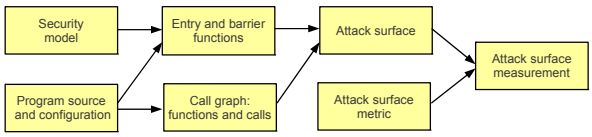


**Figure 1. Dependencies between notions defined in this section.**

$E$ would typically be the interface of the program that is exposed to the attacker, whereas $X$ would typically be the set of functions that perform authorization for privileges that the attacker is not assumed to have in the security model (e.g., administrator privileges).

**Definition 3** (Attack Surface)**.** Given a call graph $G = (F,C)$, a set of entry functions $E \subseteq F$ and a set of barrier functions $X \subseteq F$, let $G'$ be the subgraph of $G$ induced by the nodes $F' = F \setminus X$, and let $E' = E \setminus X$. The *attack surface* is then the subgraph $G_{AS}$ of $G'$ induced by all nodes $f \in F'$ such that there exists $e \in E'$ and a directed path from $e$ to $f$. By abusing notation, we denote $G_{AS} = (G,E,X)$.

The rationale behind this definition is that for most types of kernel vulnerabilities due to defects in the source code, the attacker needs to trigger the function containing the vulnerability through a call to an entry function (which, for local attackers, would be a system call). For example: for exploiting a double-free vulnerability, the attacker will need to provoke the extraneous free; for exploiting a stack- or heap-based buffer overflow, the function writing to the buffer will be reachable to the attacker; for exploiting a user-pointer dereference vulnerability, the attacker owning the user-space process will often provoke the dereference through the system call interface.

Therefore, the attack surface represents the set of functions that an attacker can potentially take advantage of.

## 2.2 Security Models

Quantifying a program's security without specifying a security model is attractive because it provides an "absolute value" to compare other programs to. However, taking into account a security model, and more generally the actual use of the program, can only result in security metrics that reflect the system's security better. As a simple example, it is common practice to measure a kernel's security by the total SLOC. However, the source code will often contain branches that will never be compiled such as architecture-specific code for other architectures. Hence, limiting the SLOC by excluding unused architecture-specific code, because this code cannot be exercised by an attacker, is already an improvement in precision.

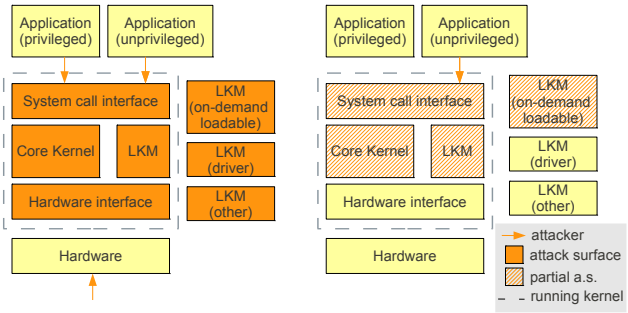


**Figure 2. On the left, the GENSEC model. On the right, the ISOLSEC model.**

We now consider the case of the Linux kernel. First, we define a generic security model that covers the dependability of the entire running kernel, and then a more specific model covering local attacks from unprivileged user space directed against the kernel. They are depicted in Figure 2.

In both cases, the hardware and the compile-time configuration of the kernel are fixed and taken into account. In both cases, the high-level security goal is to provide the traditional confidentiality, integrity and availability guarantees for the kernel: for instance, an attacker could target full control with arbitrary code execution in kernel mode, or more limited attacks such as information leakage (e.g., recover uninitialized kernel memory content) to breach confidentiality, and denial of service by crashing the kernel to reduce the system's availability. In addition, we assume that the hardware and the firmware the system is running on are trusted.

### 2.2.1 Generic Model GENSEC

The GENSEC model covers all possible kernel failures, to obtain an attack surface that is similar to the notion of TCB used for measuring security in prior work (e.g., [19, 34]).

More precisely, the attacker is both local and remote, i.e., it has an account on the target system, but can also interact with all hardware devices (e.g., sending layer-1 traffic to network interface cards). We also assume that the attacker has some amount of control over a privileged application. This means the model includes failures due to defects in the kernel in code paths that are only accessibly from a privileged application.

Therefore, in this model, a defect in any part of the running kernel — including the core kernel and all loaded LKMs, as well as any LKM that might be loaded in the future, e.g., when new hardware is plugged in — can cause a failure.

This security model may not seem intuitive, but corresponds to what is implicitly assumed when considering

the entire compiled kernel included in the TCB, a common practice.

**GENSEC attack surface** In the GENSEC model above, the attack surface is composed of the entire running kernel, as well as LKMs that can be loaded. Hence, the barrier functions set $X$ is empty, and all entry points of the kernel are included in $E$ (both hardware interrupts and system calls, as well as kernel initialization code).

#### 2.2.2 Isolation Model ISOLSEC

The ISOLSEC model reflects a common model in multi-user systems and in systems implementing defense in depth, where it is assumed an attacker has local access, e.g, by compromising an unprivileged isolated (or sandboxed) process on the system, and aims to escape the isolation by directly targeting the kernel. In this model, the attacker is malicious and has unprivileged local access, therefore it can exercise the system call interface, but not all code paths: for instance, the attacker cannot make the system call for the insertion of a new kernel module. We will detail below, when evaluating the attack surface, exactly which barrier functions should be considered.

We also assume that the attacker can target code in LKMs, including LKMs that are loaded on-demand by the system. As the attacker is not able to plug hardware into the target system, we assume that bugs in LKMs not related to installed hardware cannot lead to failures.

**ISOLSEC attack surface** An attacker in the ISOLSEC model has the set of all system calls as entry points $E$. The set of barrier functions $X$ contains functions that are only accessible from privileged applications and LKMs that cannot possibly be loaded by an action triggered by the attacker. We provide a more detailed description of those functions in the next three paragraphs.

Functions that are not reachable because of lacking permissions are highly dependent on the isolation technology used (e.g., LSM-based [11, 18, 32], chroot, LXC [28], seccomp [17]) and the policies applied to the application, and at a first approximation, we only consider the default privilege checking in use in the Linux kernel: POSIX capabilities. Hence, we assume that the set of barrier functions $X$ includes those functions performing POSIX capability checks (functions calling the `capable()` function).

However, this is not sufficient. Linux proposes a variety of pseudo-filesystems, namely `sysfs`, `debugfs`, `securityfs` and `procfs`, in which filesystem operations are dispatched to specific code paths in the kernel, mostly in LKMs, and are often used to expose information or fine-tuning interfaces to user-space processes which, in general, are privileged. However, these privilege checks are performed at the virtual filesystem layer, using POSIX ACLs: hence, they do not contain calls to the `capable()` function, and need to be considered separately. In addition, as those filesystems should not be accessible from an unprivileged application that is sandboxed (e.g., this is the case even with a simple `chroot` jail), we include all functionality provided by those four pseudo-filesystems as barrier functions $X$.

Finally, as a consequence of our assumptions on LKMs in the ISOLSEC model, we include in $X$ all LKMs that are either (*a*) not loaded while the workload is running, but not loadable on demand, or (*b*) a hardware driver that is not loaded while the workload is running.

For these reasons, we mark in Figure 2 the kernel components which can contain funcions in the attack surface only as "partial a.s (attack surface)": their inclusion depends on being reachable, after consideration of the barrier functions.

### 2.3 Attack Surface Measurements

To quantify security improvements in terms of the attack surface, we need a metric that reflects its size. Although we are not the first to make this observation [20, 30], we propose the first approach that quantifies the attack surface within a particular security model by using call graphs. In the following, we present a general approach to measure an attack surface in a security model as well as specific metrics that we will use in the case of the Linux kernel.

**Definition 4** (Code-quality metric)**.** A *code-quality metric* $\mu$ is a mapping associating a non-negative value to the nodes of the call graph:

$$\mu : F \to \mathbb{R}^+$$

**Example.** A function's SLOC (denoted *SLOC*), the cyclomatic complexity [33] (denoted *cycl*), or a CVE-based metric associating the value 1 to a function that had a CVEs in the past 7 years, and 0 otherwise (denoted $CVE$), are code-quality metrics that we use in this paper.

CVE-based metrics provide *a posteriori* knowledge on vulnerable functions: they allow an estimate of the number of CVEs a partical attack surface reduction method would have avoided, in the past. However, this metric, alone, is unsatisfactory for multiple reasons. For instance, CVEs only form a sample of all vulnerabilities existing in an application, and this sample is likely to be biased: vulnerabilities tend to be searched and discovered non-uniformly accross the code base, with often-used parts being more likely to be tested and audited. Additionally, past CVEs are not necessarily a good indicator of future CVEs: although a function with a history of vulnerabilities might be prone to more vulnerabilities in the future (e.g., due to sloppy coding style), the opposite is also likely, since this might indicate that the

function has now been thoroughly audited. For this reason, we also use *a priori* metrics such as lines of code and cyclomatic complexity, which, although imperfect for predicting vulnerabilities, do not suffer from the aforementioned issues and can be easily collected through static analysis.

**Definition 5** (Attack Surface Metric)**.** An *attack surface metric* associated with a code-quality metric $\mu$ assigns a non-negative real value to an attack surface:

$$AS_\mu : \mathscr{G} \to \mathbb{R}^+$$
$$G_{AS} \mapsto AS_\mu(G_{AS})$$

and satisfies the property:

$$\forall E' \subseteq E, \forall X' \supseteq X,\ AS_\mu(G'_{AS}) \leq AS_\mu(G_{AS})$$
$$\text{with} \quad G_{AS} = (G,E,X), G'_{AS} = (G,E',X')$$

That is, the more entry points, the higher the attack surface measurement; the more barrier functions, the lower the attack surface measurement.

**Lemma 1.** *Let m be a mapping:*

$$m : \mathscr{G} \to \mathbb{R}$$
$$G \mapsto m(G)$$

*If m satisfies:*

$$\forall G \in \mathscr{G}, \quad m(G) \geq 0$$
$$\forall G' \subseteq G \in \mathscr{G}, \quad m(G') \leq m(G)$$

*then it is an attack surface metric.*

*Proof.* Let $G_{AS} = (G,E,X), G'_{AS} = (G,E',X')$ such that $E' \subseteq E$ and $X' \supseteq X$. Then:

$$G'_{AS} \subseteq G_{AS}$$

Hence $m$ satisfies the property in Definition 5:

$$m(G'_{AS}) \leq m(G_{AS})$$

□

Note that this property is not necessary to satisfy Definition 5, because a smaller set of functions (in $G'_{AS}$) should not necessarily mean a smaller attack surface measurement. This is sensible, because in practice some functions can reduce the overall attack surface (e.g., by sanitizing input), and an attack surface metric could take this into account (e.g., Murray, Milos, and Hand [36] propose such a metric for measuring TCB size). Such an example is depicted in Figure 1: A metric satisfying Lemma 1 would always measure a lower attack surface for $G'_{AS}$ than for $G_{AS}$, whereas this is not necessary for a metric satisfying Definition 5.

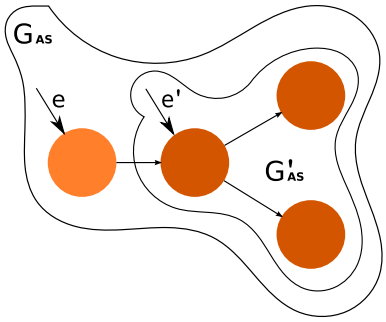


**Figure 3. Example attack surfaces $G_{AS}$ (with $E = \{e\}$ and $X = \emptyset$) and $G'_{AS}$ (with $E' = \{e'\}$ and $X' = \emptyset$). Note that $E' \not\subseteq E$ and $G'_{AS} \subseteq G_{AS}$.**

**Proposition 1.** *The following two functions are attack surface metrics:*

$$AS1_\mu(G_{AS}) = \sum_{i \in F_{AS}} \mu(i)$$

$$AS2_\mu(G_{AS}) = {\mu_{AS}}^T L(\widetilde{G_{AS}}) \mu_{AS}$$

*where* $G_{AS} = (F_{AS}, C_{AS})$, ${\mu_{AS}}^T = (\mu(1), \ldots, \mu(|F|))$, *and* $L(G)$ *is the Laplacian matrix of a simple (non-directed) graph:*

$$L(G) = D - A$$

*where D is a diagonal matrix with the degrees of the nodes on the diagonal, and A the adjacency matrix of the graph* ($A_{ij} = 1$ *when the (i,j) edge exists, 0 otherwise). As* $G_{AS}$ *is directed, we transform it into a simple graph by ignoring the direction on its arcs, which we denote* $\widetilde{G_{AS}}$.

$AS1$ provides a simple and intuitive formulation of an attack surface metric: for instance, $AS1_{SLOC}$ is a sum of the lines of code in the attack surface. However, it values each function equally. $AS2$ takes advantage of the functions position in the call graph, and attaches more value to code-quality metrics in functions that have a large number of callers (and callees) that have a lower code-quality measurement. The Appendix contains a proof, and a more detailed explanation of the formulation of $AS2$. We use both these attack surface metrics in our evaluations in Section 4.

## 2.4 Summary

The metrics introduced in this section are for the purpose of a precise evaluation of the security gains of our approach. These metrics contain metrics used commonly in prior work, such as total TCB size in SLOC ($AS1_{SLOC}$ in the GENSEC model). We do not claim the metrics presented in this section are the panacea in measuring attack surfaces. Rather, we propose new metrics that take into account what

attackers are capable of. This will allow us to discuss attack surface reduction results in Section 5 for additional insights into the advantages and disadvantages of tailoring the Linux kernel configuration.

# 3 Kernel Tailoring

## 3.1 General Idea and Solution Overview

The formalism introduced in Section 2 provides a solid means to calculate the attack surface in a given security model. We apply these theoretical considerations to improve the overall system security of Linux as shipped by Linux distributions such as Ubuntu or Red Hat Linux. These popular distributions cannot afford to ship and maintain a large number of different kernels. Therefore, they configure their kernels to be as generally usable as possible, which requires the kernel package maintainers responsible to enable as much functionality (i.e., KCONFIG features) as possible. Unfortunately, this also maximizes the attack surface. As security-sensitive systems do not require the genericalness provided, the attack surface can be reduced by simply not enabling unnecessary features. What features are necessary, however, depends on the actual workload of the corresponding use-case. Therefore, our approach consists of two phases. In the analysis phase, the workload is analyzed at run time. The second phase calculates a reduced Linux configuration that enables only the functionality that has been observed in the analysis phase.

## 3.2 Configuration Mechanisms in Linux

Variability in Linux is centrally managed by means of KCONFIG, which is both a tooling and a configuration language, in which constraints such as dependencies and conflicts are modeled in a domain specific language (DSL). In the literature, the formal semantics [44, 55] have been analyzed for use in variability extractors [52], which translate the specified constraints into propositional formulas. These formulas are essential for constructing the optimized Linux configuration.

However, the implementation of variability is very scattered in Linux, which makes holistic reasoning challenging. In practice, the analysis of KCONFIG files, MAKE files and C Preprocessor (`CPP`) code requires very specialized and sophisticated extraction tools. A reliable mapping of user-configurable features in KCONFIG to source lines in the source tree requires a correct combination of all three sources of variability. A solid understanding of the Linux build system KBUILD and the configuration tool KCONFIG is instrumental to correctly relate the variability declaration in KCONFIG. This work therefore implements our approach as extension to existing work [13, 37, 52], which has been kindly provided to us by the original authors.

## 3.3 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 steps necessary are shown in Figure 4.

❶ **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 code that can be evaluated at runtime.

In addition, our approach requires a kernel that is 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`.

To also cover code that is executed at boot time by initialization scripts, we need to enable the `ftrace` as early as possible. For this reason, we modify the initial RAM disk, which contains programs and LKMs for low-level system initialization [2]. Linux distributions use this part of the boot process to detect installed hardware early in the boot process and, mostly for performance reasons, load only the required essential device drivers. This basically turns on tracing even before the first process (`init`) starts.

❷ **Run workload.** In this step, the system administrator runs the targeted application or system services. 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.

During this run, we copy the output of the tracing wrapper script at constant time intervals. This allows us to compare at what time what functionality was accessed, and

[2] This part of the Linux plumbing is often referred to as "early userspace"

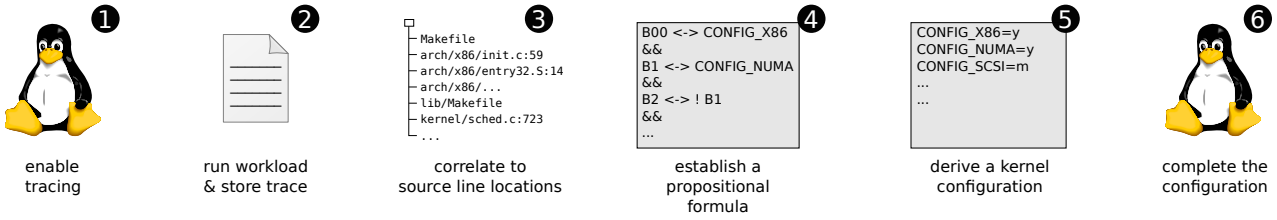


**Figure 4. Kernel-Configuration Tailoring Workflow**

therefore to monitor the evolution of the tailored kernel configuration over time based on these snapshots.

❸ **Correlation to source lines.** A system service translates the raw address offsets into source line locations using the ADDR2LINE tool from the `binutils` tool suite. Because LKMs are relocated in memory depending on their non-deterministic order of loading, the system service compares the raw, traced addresses to offsets in the LKM's code segment. This allows the detection of functionality that is not compiled statically into the Linux kernel. This correlation of absolute addresses in the code segment with the debug symbols allows us to identify the source files and the `#ifdef` blocks that are actually being executed during the tracing phase.

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 `CPP` block, KCONFIG item and source file can appear as propositional variable in the resulting formula. This formula is constructed with the variability constraints extracted from `#ifdef` blocks, KCONFIG feature descriptions and Linux Makefiles. The extractors we use have been developed, described and evaluated in previous work [13, 46, 52]. 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 **Completing the Linux kernel configuration.** The resulting kernel configuration now contains all features that have been observed in the analysis phase. The caveat is that the resulting propositional formula can only cover KCONFIG features of code that has been traced. In principle, features that are left unreferenced are to be deselected. However, features in KCONFIG declare non-trivial dependency constraints [55], which must all hold for a given configuration in order to produce a *valid* KCONFIG configuration. The problem of finding a feature selection with the smallest number of enabled features, (which is generally not unique) has the complexity *NP-hard*. We therefore rely on heuristics to find a sufficiently small configuration that satisfies all constraints of KCONFIG but is still significantly smaller compared to a generic distribution kernel.

## 4 Evaluation

In this section, we present two use cases, namely a LAMP-based server and a graphical workstation that provides an NFS service, both on distinct, non-virtualized hardware, that we use to evaluate the effects of kernel-configuration tailoring. This evaluation demonstrates the approach with practical examples, verifies that the obtained kernel is functional, i.e., no required configuration option is missing in the tailored kernel, and shows that the performance of the kernel with the configuration generated remains comparable to that of the distribution kernel. We quantify the attack surface reduction achieved with the formalisms described in Section 2.

### 4.1 Overview

In both use cases, we follow the process described in Section 3 to produce a kernel configuration that is tailored to the respective use case. For each use case, we detail the workload that is run to collect traces in the following subsections. Both machines use the 3.2.0-26 Linux kernel distributed by Ubuntu as baseline, which is the kernel shipped at the time of this evaluation in Ubuntu 12.04.

To compare the performance, we use benchmarks that are specific to the use case. We repeat both experiments at least 10 times and show 95%-confidence intervals in our fig-

ures where applicable. The benchmarks compare the original, distribution-provided kernel to the tailored kernel generated. All requests are initiated from a separate machine over a gigabit Ethernet link. To avoid interferences by start-up and caching effects right after the system boots, we start our workload and measurements after a warm-up phase of 5 min.

To measure the attack surface reduction, we first calculate code-quality metrics for each function in the kernel by integrating the FRAMA-C [15] tool into the kernel build system. For CVEs, we parse all entries for the Linux kernel in the National Vulnerability Database (NVD)[3]. For entries with a reference to the GIT repository commit (only those CVEs published after 2005), we identify the C functions that have been changed to patch a security issue, and add each function to a list. Our metric assigns a value of 1 to functions that are in this list, and 0 otherwise. We also generate static call graphs for each use case by using both FRAMA-C and NCC [38] and combining both call graphs to take into account calls through function pointers, which are very widely used in the Linux kernel. In the case of the GENSEC model, we compute the *AS*1 and *AS*2 attack surface metrics directly over all functions in this graph, for both the baseline and the tailored kernel. In the case of the ISOLSEC model, we compute the subgraph corresponding to the attack surface by performing a reachability analysis from functions corresponding to system calls (entry points) and removing all barrier functions as detailed in Section 2.2.2. We then evaluate the security improvements by computing the attack surface reduction between the baseline kernel and a tailored kernel.

## 4.2 LAMP-stack use case

### 4.2.1 Description

This use case employs a machine with a 2.8 GHz Celeron CPU and 1 GB of RAM. We use the Ubuntu 12.04 Server Edition with all current updates and no modifications to either the kernel or any of the installed packages. As described in Section 3.3, we extend the system-provided initial RAM disk (`initrd`) to enable tracing very early in the boot process. In addition, we set up an web platform consisting of APACHE2, MYSQL and PHP. The system serves static documents, the collaboration platform DOKUWIKI [16] and the message board system PHPBB3 [40] to simulate a realistic use case.

The test workload for this use case starts with a simple HTTP request using the tool WGET, which fetches a file from the server right after the five-minute warm-up phase. This is followed by one run of the HTTPERF [35] tool, which accesses a static website continuously, increasing the number of requests per second for every run. Finally, we run the SKIPFISH [54] security scan on the server. SKIPFISH is a tool performing automated security checks on web applications, hence exercising a number of edge-cases, which is valuable not only to exercise as many code paths as possible, but also to test the stability of the tailored use case.

[3] http://nvd.nist.gov/

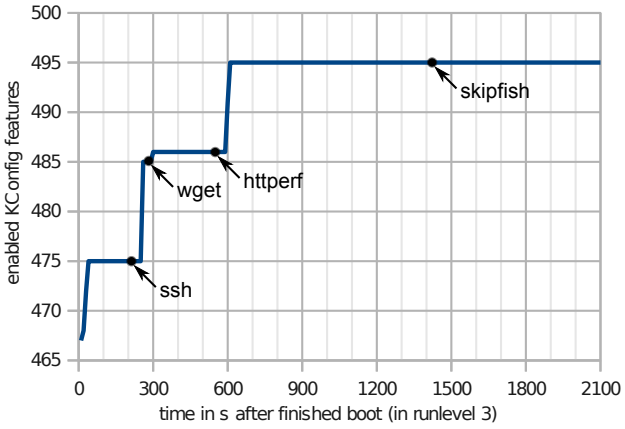


**Figure 5. Evolution of KCONFIG features enabled over time. The bullets mark the point in time at which a specific workload was started.**

### 4.2.2 Results

Figure 5 depicts the number of KCONFIG features that our tool obtains from the trace logs collected at the times given. After the warm-up phase, connecting to the server via `ssh` causes a first increase in enabled KCONFIG features. The simple HTTP request triggers only a small further increase, and the configuration converges quickly after the HTTPERF tool is run, and shows no further changes when proceeding to the SKIPFISH scan. This shows that, for the LAMP use case, a tracing phase of about ten minutes is sufficient to detect all required features.

**Tailoring** The trace file upon which the kernel configuration is generated is taken 1,000 sec after boot, i.e., after running the tool HTTPERF, but before running the SKIPFISH tool. It consists of 8,320 unique function addresses, including 195 addresses from LKMs. This correlates to 7,871 different source lines in 536 files. Our prototype generates the corresponding configuration in 145 seconds and compiles the kernel in 89 seconds on a commodity quad-core machine with 8 GB of RAM.

When comparing the original kernel to the distribution kernel shipped with Ubuntu, we observe a reduction of KCONFIG features that are statically compiled into the kernel of over 70%, and almost 99% for features that lead to

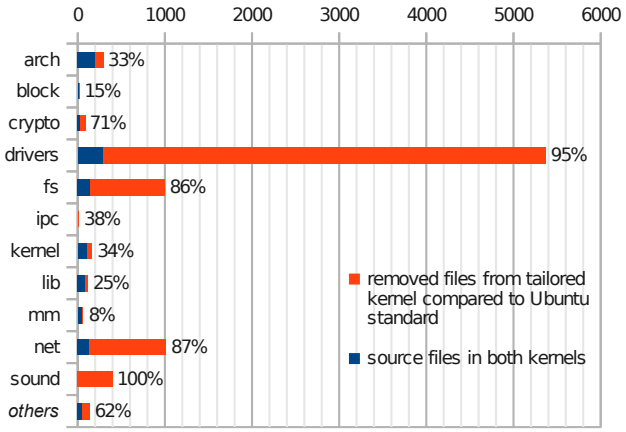


**Figure 6. Reduction in compiled source files for the tailored kernel, compared with the baseline in the LAMP use case (results for the workstation with NFS use case are similar). For every subdirectory in the Linux tree, the number of source files compiled in the tailored kernel is depicted in blue and the remainder to the number in the baseline kernel in red. The reduction percentage per subdirectory is also shown.**

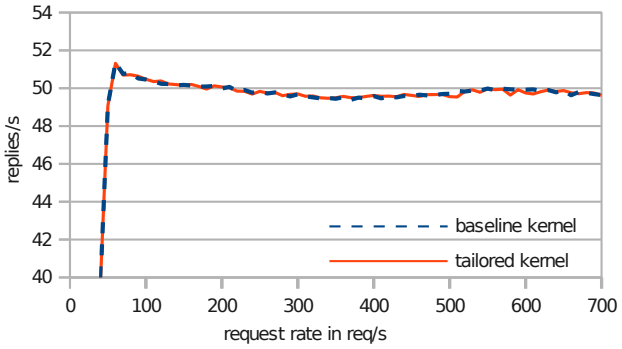


**Figure 7. Comparison of reply rates of the LAMP-based server using the kernel shipped with Ubuntu and our tailored kernel. Confidence intervals were omitted, as they were too small and thus detrimental to readability.**

compilation as LKMs (cf. Table 1). Consequently, the overall size of the text segment for the tailored kernel is over 90% lower than that of the baseline kernel supplied by the distribution.

To relate to the savings in terms of attack surface, we show the number of source code files that the tailored configuration does not include when compared to the distribution configuration in Figure 6. The figure breaks down the reduction of functionality by subdirectories in terms of source files that get compiled. The highest reduction rates are observed inside the `sound/` (100%), `drivers/` (95%), and `net/` (87%) directories. As the web server does not play any sounds, the trace file does not indicate any sound-related code. Similarly, the majority of drivers are not needed for a particular hardware setup. The same applies to most of the network protocols available in Linux, which are not required for this use case. Out of 8,670 source files compiled in the standard Ubuntu distribution kernel, the tailored kernel only required 1,121, which results in an overall reduction of 87% (cf. Table 1).

**Stability** To ensure that our tailored kernel is fully functional, we run SKIPFISH [54] once on the baseline kernel and then compare the results to a scan on the tailored kernel. The report produced by the tool finds no significant difference from one kernel configuration to the other, hence the tailored kernel can handle unusual web requests equally well. Furthermore, this shows that for this use case even a kernel tailored from a trace file which only covers a smaller test workload than the target scenario is suitable for stable operation of the service.

**Performance** We measure the performance with the HTTPERF tool. The result is compared with a run performed on the same system that runs the baseline kernel. Figure 7 shows that the tailored kernel achieves a performance very similar to that of the kernel provided by the distribution.

**Security** Finally, we compute attack surface reduction with *AS*1 and *AS*2 in the GENSEC and ISOLSEC models after generating the relevant call graphs. The numbers in Table 1 show that the $AS1_{SLOC}$, $AS1_{cycl}$ and $AS2_{SLOC}$ attack surface reduction is around 85% in the GENSEC model, and around 80% in the ISOLSEC model. In both models, there are also 60% fewer functions that were affected by patches due to CVEs in the past. We also observe that $AS2_{cycl}$ is slightly lower, with an attack surface reduction around 60%. Overall, the attack surface reduction is between 60% and 85%.

## 4.3 Workstation/NFS use case

### 4.3.1 Description

For the workstation/NFS server use case, we use a machine with a 3.4 GHz quad-core CPU and 8 GB of RAM, running the Ubuntu 12.04 Desktop edition, again without modifications to packages or kernel configuration. The machine is configured to export a local directory via NFS.

| | | Baseline | | Tailored | | Reduction | |
|---|---|---|---|---|---|---|---|
| | | LAMP | NFS | LAMP | NFS | LAMP | NFS |
| Kernel (vmlinux) size in Bytes | | | 9,933,860 | 4,228,235 | 4,792,508 | 56% | 52% |
| LKM total size in Bytes | | | 62,987,539 | 2,139,642 | 2,648,034 | 97% | 96% |
| Options set to 'y' | | | 1,537 | 452 | 492 | 71% | 68% |
| Options set to 'm' | | | 3,142 | 43 | 63 | 99% | 98% |
| Compiled source files | | | 8,670 | 1,121 | 1,423 | 87% | 84% |
| GENSEC | Call graph nodes | | 230,916 | 34,880 | 47,130 | 85% | 80% |
| | Call graph arcs | | 1,033,113 | 132,030 | 178,523 | 87% | 83% |
| | $AS1_{SLOC}$ | | 6,080,858 | 895,513 | 1,122,545 | 85% | 82% |
| | $AS1_{cycl}$ | | 1,268,551 | 209,002 | 260,189 | 84% | 79% |
| | $AS1_{CVE}$ | | 848 | 338 | 429 | 60% | 49% |
| | $AS2_{SLOC}$ | | 58,353,938,861 | 11,067,605,244 | 11,578,373,245 | 81% | 80% |
| | $AS2_{cycl}$ | | 2,721,526,295 | 1,005,337,180 | 1,036,833,959 | 63% | 62% |
| | $AS2_{CVE}$ | | 20,023 | 7,697 | 9,512 | 62% | 52% |
| ISOLSEC | Call graph nodes | 92,244 | 96,064 | 15,575 | 21,561 | 83% | 78% |
| | Call graph arcs | 443,296 | 462,433 | 64,517 | 89,175 | 85% | 81% |
| | $AS1_{SLOC}$ | 2,403,022 | 2,465,202 | 425,361 | 550,669 | 82% | 78% |
| | $AS1_{cycl}$ | 504,019 | 518,823 | 99,674 | 126,710 | 80% | 76% |
| | $AS1_{CVE}$ | 485 | 524 | 203 | 276 | 57% | 47% |
| | $AS2_{SLOC}$ | 15,753,006,783 | 15,883,981,161 | 4,457,696,135 | 4,770,441,587 | 72% | 70% |
| | $AS2_{cycl}$ | 918,429,105 | 929,197,559 | 374,455,910 | 391,855,241 | 59% | 57% |
| | $AS2_{CVE}$ | 10,151 | 11,127 | 4,287 | 5,489 | 57% | 51% |

**Table 1. Summary of kernel tailoring and attack surface measurements.**

To measure the performance of the different kernel versions, we use the BONNIE++ [10] benchmark, which covers reading and writing to this directory over the network. To achieve results that are meaningful, we disable caching on both server and client.

#### 4.3.2 Results

The trace file of the configuration selected for further testing consists of 13,841 lines that reference a total of 3,477 addresses in modules. This resolves to 13,000 distinct source lines in 735 files. Building the formula and therefore the configuration takes 219 seconds, compiling the kernel another 99 seconds on the same machine as described above. We observe a reduction of KCONFIG features that are statically compiled into the kernel by 68%, 98% for features compiled into LKMs, and about 90% less code in the text segment.

**Performance and Stability** We did not find any impact on the regular functionality of the workstation, i.e., all hardware attached, such as input devices, Ethernet or sound, remained fully operable with the tailored kernel booted. Using the tailored kernel, we run BONNIE++ again with the same parameters, and compare the results with those of the distribution kernel. Figure 8 shows that also in this use case the kernel compiled with our tailored configuration achieves a very similar performance.

**Security** Attack surface reduction results are similar to the LAMP use case. The numbers in Table 1 show that the $AS1_{SLOC}$, $AS1_{cycl}$ and $AS2_{SLOC}$ attack surface reduction is around 80% in the GENSEC model, and around 75% in the ISOLSEC model. In both models, there are also 50% fewer functions that were affected by patches due to CVEs in the past. We also observe that $AS2_{cycl}$ is slightly lower as well, with attack surface reduction around 60%. Overall, our measurements suggest the attack surface reduction is between 50% and 80%.

## 5 Discussion

### 5.1 Attack surface measurements

This section discusses the results of our attack surface measurements.

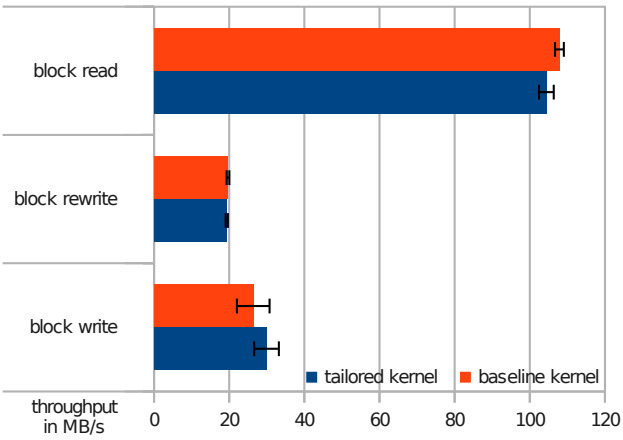


**Figure 8. Comparison of the test results from the BONNIE++ benchmark, showing no significant difference between the tailored and the baseline kernel.**

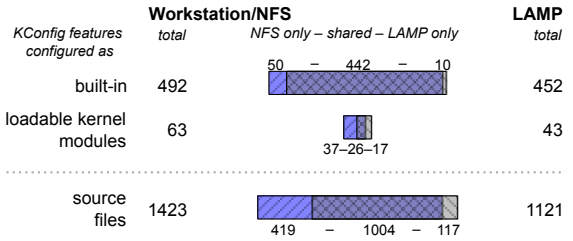


**Figure 9. Comparison of the two generated configurations from the use cases in terms of KCONFIG features leading to built-in code and code being compiled as LKM. Below, the total number of compiled source files is compared between the two resulting kernels.**

**Use cases** Figure 9 shows that the tailored kernel configurations are largely similar for both cases. We observe a number of features that differentiate the use cases, both in terms of hardware and workload. The workstation/NFS use case requires the highest number of differentiating features (87 enabled KCONFIG options for NFS compared to 27 for LAMP). This can be explained by the setup (the desktop version of Ubuntu has the X11 window system installed and running, whereas the server version has not) and by the workload: as NFS also runs in kernel mode, additional kernel features are required. This point is useful for understanding attack surface reduction results. Although both use cases show similar $AS1_{SLOC}$ reductions (around 80%), there is a slight difference for both GENSEC and ISOLSEC and the various *AS* metrics in the reduction achieved in favor of the LAMP use case (see Table 1). This is simply because the workstation/NFS use case requires a larger kernel than the LAMP one.

**The case of CVE-2010-3904** Out of the 422 CVEs we have inspected, we detail the case of one highly publicized vulnerability for illustration purposes. CVE-2010-3904 documents a vulnerability that is due to a lack of verification of user-provided pointer values, in RDS, a rarely used socket type. An exploit for obtaining local privilege escalation was released in 2010 [42]. We verified that in the case of the workstation/NFS use case, both tailored kernel configurations have the functionality removed in the GENSEC and ISOLSEC models, and thus would have prevented the security issue. In contrast, the baseline kernel contains the previously-vulnerable feature in the GENSEC and ISOLSEC models.

**CVE sampling bias** The results in Table 1 show slightly lower CVE reduction numbers than for all other metrics, especially in the case of *AS*1. We hypothesize that this small difference is due to a sampling bias: code that is used more often is also audited more often, more bug reports concerning it are submitted, and better care is taken in documenting the vulnerabilities of such functions. We also observe the average number of CVEs per function is lower in the functions that are in the tailored kernel, when compared to those functions that are not. Previous studies [9, 39] have shown that code in the `drivers/` sub-directory of the kernel, which is known to contain a significant amount of rarely used code, on average contains significantly more bugs than any other part 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 attack surface reduction as presented in this paper.

Nevertheless, we still take the CVE reduction numbers into account, because they reflect a posteriori knowledge about vulnerability occurrences. All our measurements indicate attack surface reduction lies approximately within 50% and 85% across all parameters (use cases, security models, metrics), which is a very positive result for kernel tailoring.

**Attack surface metric comparison** The *AS*1 and *AS*2 results are quite close, which, considering how different their formulations are, shows the robustness of the simple attack surface definition introduced in Section 2. *AS*2 is also of interest because it introduces the use of the Laplacian, which is instrumental in many applications of graph theory (e.g., for data mining [2]), for the purpose of attack surface measurements.

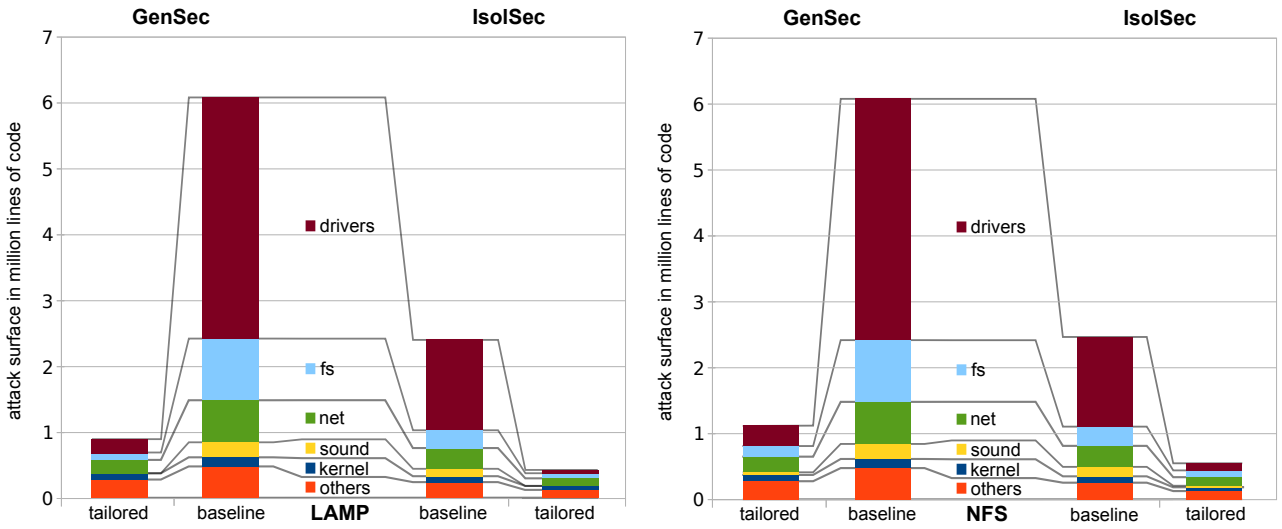


**Figure 10.** $AS1_{SLOC}$ **attack surface measurements per kernel subsystem in both security models and use cases.**

**Comparison to kernel extension isolation** Approaches such as [31, 50] provide a way, through impressive technical feats, of isolating LKMs from the kernel, i.e., running them with lesser privileges. This means, ideally, the compromise of an LKM by an attacker cannot lead to kernel compromise. To evaluate how such solutions compare to kernel tailoring, we again make use of the attack surface formalism introduced in Section 2. Assuming that these isolation solutions are ideal (i.e., that their own implementation does not increase the kernel's attack surface and the attackers are not able to bypass the isolation), we remove all LKMs from the baseline kernel's attack surface in the IsolSec model, hence obtaining a lower bound of the real attack surface of such LKM-isolated kernels. Our results in Table 2 show that kernel tailoring is superior to LKM isolation: for instance, the $AS1_{SLOC}$ measurement of the ideal LKM isolation is four times higher. We also evaluate whether combining both approaches could be beneficial, i.e., first generating a tailored kernel and then applying an ideal LKM isolation. The results show that the resulting attack surface is not significantly lower than that obtained by kernel tailoring alone, which further confirms the improvements of our approach, even when compared to an ideal LKM isolation solution. Additionally, we remark that this lower bound is also applicable to approaches that prevent automatic-loading of LKMs, such as the well-known grsecurity kernel patch with the `MODHARDEN` option [48].

**Security models** The attack surface reduction is important in both security models, but more so in the GenSec model. This can be attributed to the fact that the GenSec model includes a large number of drivers, whereas the IsolSec model does less. As can be seen from Figure 10, the attack surface reduction is particularly high for drivers. In other words, tailoring appears to be slightly more effective in the GenSec model than in the IsolSec model. This is to be expected, since our approach reduces the kernel's attack surface system-wide (and not per-process). Figure 10 also shows that, both in the baseline and tailored kernels and independently of the use case, the IsolSec attack surface is about half of the GenSec attack surface. In other words, the attack surface of a local attacker (as defined in the IsolSec model) is about half of what is generally considered as the TCB of the kernel.

**Importance of kernel configuration** When quoting SLOC measurements of the Linux kernel as a simple way of quantifying TCB size, we advocate specifying the kernel configuration the measurement corresponds to. Indeed, our results show that, depending on the kernel configuration, the total number of lines of code can vary by up to an order of magnitude. An other important factor is the kernel version, since the Linux kernel increased significantly in size over the past years.

## 5.2 Kernel tailoring

We will discuss now the key strengths and weaknesses of the kernel-tailoring tool with respect to various properties.

**Effectiveness** Although in absolute terms the attack surface of the tailored Linux kernel remains high (for $AS1$,

| | Ideal LKM isolation | Kernel Tailoring | | Both combined | |
|---|---|---|---|---|---|
| | | LAMP | Workstation/NFS | LAMP | Workstation/NFS |
| $AS1_{SLOC}$ | 2,064,526 | 425,361 | 550,669 | 420,373 | 489,732 |
| $AS1_{cycl}$ | 444,775 | 99,674 | 126,710 | 98,534 | 113,735 |
| $AS1_{CVE}$ | 390 | 203 | 276 | 203 | 240 |
| $AS2_{SLOC}$ | 11,826,476,219 | 4,457,696,135 | 4,770,441,587 | 4,452,329,879 | 4,663,745,009 |
| $AS2_{cycl}$ | 851,676,457 | 374,455,910 | 391,855,241 | 374,214,950 | 386,472,434 |
| $AS2_{CVE}$ | 7,725 | 4,287 | 5,489 | 4,287 | 4,849 |

**Table 2. Comparison of ISOLSEC attack surface measurements between an ideal LKM isolation approach (a lower bound of the attack surface of kernel extension fault isolation approaches) and our approach, when applied to the current Ubuntu 12.04 Kernel. The third column represents attack surface measurements that would result if both approaches were combined.**

about 500K SLOC in the ISOLSEC model, and 1000K SLOC in the GENSEC model), Table 1 shows that for both use cases and across all meaningful metrics, the attack surface is reduced by almost an order of magnitude. As such, vulnerabilities existing in the Linux kernel sources are significantly less likely to impact users of a tailored kernel. This makes the approach presented an effective means for improving security in various use cases.

**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 which kernel functionalities the application requires and therefore, which kernel configuration options have to be enabled. With the increasing importance of compute clouds, where customers use virtual machines for very dedicated services such as the LAMP stack presented in Section 4, we expect that our approach will prove valuable for improving the security in many cloud deployments.

**Usability** Most of the steps presented in Section 3 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 constantly receives maintenance in the form of bug fixes and security updates from the Linux distributor. We therefore are confident that our approach to tailor a kernel configuration for specific use-cases automatically is both practical and feasible to implement in real-world scenarios.

**Extensibility** The experiments in Section 4 show that, for proper operation, the resulting kernel requires eight additional KCONFIG options, which the `ftrace` feature could not detect. By using a whitelist mechanism, 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.

**Safety** Many previous approaches that reduce the Linux kernel's TCB (e.g., [17], [24]) introduce additional security infrastructure in form of code that prevents functionality in the kernel from being executed, which can lead to unexpected impacts and the introduction of new defects into the kernel. In contrast, our approach modifies the kernel configuration instead of changing the kernel sources (e.g., [25, 48]) or modifying the build process (e.g., [12]). In that sense, our approach, by design, cannot introduce new defects into the kernel.

However, as the configurations produced are specific to the use case analyzed in the tracing phase, we cannot rule out that the tailored configuration uncovers bugs that could not be observed in the distribution-provided Linux kernel. Although we have not encountered any of such bugs in practice, we would expect them to be rather easy to fix, and of rare occurence, as the kernels produced contain a strict subset of functionality. In some ways, our approach could therefore even help improve Linux by uncovering bugs that are hard to detect.

This also emphasizes the importance of the analysis phase, which must be sufficiently long to observe all necessary functionality. In case of a crash or similar failure, however, we could only attribute this to a bug in either the kernel or the application implementation that needs to be fixed. In other words, this approach is safe by design.

## 6 Related work

This paper is related to previous research from many areas: improving OS kernel reliability and security, reducing the attack surface of the kernel towards user-space applications, specializing kernels for embedded systems, measuring attack surfaces, and code complexity.

**Kernel specialization** Several researchers have suggested approaches to tailor the configuration of the Linux kernel, although security is usually not a goal. Instead, most often improvements in code size or execution speed are targeted. For instance, Lee et al. [25] 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. [8], in contrast, propose a method based on link-time binary rewriting, and 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 and it is significantly safer, because we do not make any unsupported changes to the kernel.

**Micro-kernel architectures and retrofitting security** TCB size reduction has always been a major design goal for micro-kernels [1, 27], and in turn facilitates a formal verification of the kernel [22] or its implementation in safer languages, such as OCaml [29]. Our work achieves this goal with a widely-used monolithic kernel, i.e., Linux, without the need of new languages or concepts.

A number of approaches exist that retrofit micro-kernel–like features into monolithic OS kernels, mostly targeting fault isolation of kernel extensions such as device drivers [7, 31, 50]. For instance, the work of Swift et al. [50] wraps calls from device drivers to the core Linux kernel API (and vice-versa), as well as use virtual memory protection mechanisms, which leads to a more reliable kernel in the presence of faulty drivers. In the presence of a malicious attacker who can compromise such devices, however, this is in general insufficient. This can be mitigated with more involved approaches such as LXFI [31], which requires interfaces between the kernel and extensions to be annotated manually. An alternative is to prevent potential vulnerabilities in the source code from being exploitable in the first place. For instance, Secure Virtual Architecture (SVA) [12] compiles the existing kernel sources into a safe instruction set architecture, which is translated to native instructions by the SVA VM. This provides among other guarantees, a variant of type safety and control flow integrity. However, it is very difficult to recover from attacks (or false positives) without crashing the kernel with such defenses [26]. In contrast, kernel tailoring only uses the built-in configurability of Linux, hence kernel crashes can only be due to defects already present in the kernel.

**Kernel attack surface reduction** The ISOLSEC model used in this paper is commonly used when building *sandboxes* or *isolation* solutions, in which a set of processes must be contained within a particular security domain (e.g., with [11, 18, 32], which are all based on the Linux Security Module (LSM) framework [53]). As we have demonstrated, adjusting the kernel configuration also significantly reduces the attack surface in such a model (this corresponds to the ISOLSEC model). The idea of directly restricting or monitoring for intrusion detection the system call interface on a per-process basis has been extensively explored (e.g., [23, 41] and references in [14]), although not often with specific focus on reducing the kernel's attack surface (i.e., reducing $AS1_{SLOC}$ in the ISOLSEC model), or in other words, to specifically prevent vulnerabilities in the kernel from being exploited by reducing the amount of code reachable by an attacker in this model

SECCOMP [17] directly tackles this issue by allowing processes to be sandboxed at the system call interface. KTRIM [24] 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. In future work, we will investigate how dynamic approaches such as SECCOMP or KTRIM can be combined with the static tailoring of the kernel configuration most effectively.

**Analysis of variability in Linux** This work relies on static analysis to identify the implementation of variability in Linux. Berger et al. [5] statically analyze the implementation and expressiveness of the variability declaration languages of Linux and eCos, an operating systems targeted at embedded systems, with the goal to extract a reliable feature-to-code mapping. In our approach, we make use of this mapping for Linux when establishing the propositional formula from the identified source line locations in the traces (Step ❹ in Figure 4). The work of Berger et al. [5] is continued in a follow-up publication [4] and by Nadi and Holt [37], which analyze implementation anomalies in KBUILD. Unfortunately, both extractors are based on parsing MAKE files, which turns out to be error-prone or to require adaptations for each new Linux kernel version [13]. We therefore use an improved version of the GOLEM tool by Dietrich et al. [13], which extracts variability from KBUILD with a sufficient accuracy for this work.

The variability extracted the GOLEM tool is combined with the variability model used by the UNDERTAKER

tool, which checks for configuration inconsistencies in Linux [52]. Configuration inconsistencies manifest themselves in `#ifdef` blocks that are only seemingly configurable, but in fact are not in any KCONFIG configuration. While in this work we do not aim to improve the Linux implementation, we have extended the UNDERTAKER tool to generate the tailored configurations. The necessary modifications were straightforward to implement, and we will include them into the next public release.

**Complexity, security metrics, and attack surface** The need for better security metrics is widely accepted in both academia and industry [3, 21, 43, 49]. Howard, Pincus, and Wing [20] were the first to propose the use of code complexity and bug count metrics to compare the relative "attackability" of different software, and others have followed [30, 45, 47]. Murray, Milos, and Hand [36] underline the fact that TCB size measurements by SLOC, while good, might not be precise enough because additional code can sometimes reduce the attack surface (e.g., sanitizing input). Manadhata and Wing [30] present an attack surface metric based on an insightful I/O automata model of the target system, taking into account in particular the data flow from untrusted data items and the entry points of the system. The definition of attack surface used in their work closely relates to ours, with the differences that our modeling is solely based on static call graphs and a measure of code complexity of each underlying function. In contrast, this work measures the attack surface with respect to a particular attacker model.

## 7 Conclusion

Linux distributions ship "generic" kernels, which contain a considerable amount of functionality that is provided just in case. For instance, a defect in an unnecessarily provided driver may be sufficient for attackers to take advantage of. The genericalness of distribution kernels, however, is unnecessary for concrete use cases. This paper presents an approach to optimize the configuration of the Linux kernel. The result is a hardened system that is tailored to a given use case in an automated manner. We evaluate the security benefits by measuring and comparing the attack surface of the kernels that are obtained. The notion of attack surface is formally defined and evaluated in a very generic security model, as well as a security model taking precisely into account the threats posed by a local unprivileged attacker.

We apply the prototype implementation of the approach in two scenarios, a Linux, Apache, MySQL and PHP (LAMP) stack and a graphical workstation that serves data via network file system (NFS). The resulting configuration leads to a Linux kernel in which unnecessary functionality is removed at compile-time and thus, inaccessible to attackers. We evaluate this reduction using a number of different metrics, including SLOC, the cyclomatic complexity and previously reported vulnerability reports, resulting in a reduction of the attack surface between about 50% and 85%. Our evaluations also indicate that this approach reduces the attack surface of the kernel against local attackers significantly more than previous work on kernel extension isolation for Linux. We are convinced that the presented approach improves the overall system security and is practical for most use cases because of its applicability, effectiveness, ease and safety of use.

## 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.

## 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*. 1986, pages 93–113.

[2] Mikhail Belkin and Partha Niyogi. "Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering". In: *Advances in Neural Information Processing Systems 14*. 2001, pages 585–591.

[3] S.M. Bellovin. "On the Brittleness of Software and the Infeasibility of Security Metrics". In: *Security Privacy, IEEE* 4.4 (2006), page 96. ISSN: 1540-7993. DOI: 10.1109/MSP.2006.101.

[4] Thorsten Berger, Steven She, Krzysztof Czarnecki, and Andrzej Wasowski. *Feature-to-Code Mapping in Two Large Product Lines*. Technical report. University of Leipzig (Germany), University of Waterloo (Canada), IT University of Copenhagen (Denmark), 2010.

[5] Thorsten Berger, Steven She, Rafael Lotufo, and Andrzej Wasowski und Krzysztof Czarnecki. "Variability Modeling in the Real: A Perspective from the Operating Systems Domain". In: *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. (Antwerp, Belgium). 2010, pages 73–82. ISBN: 978-1-4503-0116-9. DOI: 10.1145/1858996.1859010.

[6] N. Biggs. *Algebraic Graph Theory*. 1974.

[7] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. "Fast byte-granularity software fault isolation". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP '09. 2009, pages 45–58. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629581.

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

[8] 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)*. 2005, pages 95–104. ISBN: 1-59593-018-3. DOI: 10.1145/1065910.1065925.

[9] 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). 2001, pages 73–88. ISBN: 1-58113-389-8. DOI: 10.1145/502034.502042.

[10] Russell Coker. *Bonnie++. Benchmark suite for hard drive and file system performance*. URL: http://www.coker.com.au/bonnie++/ (visited on 08/02/2012).

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

[12] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. "Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*. (Stevenson, WA, USA). 2007, pages 351–366. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294295.

[13] 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). 2012, pages 21–30. ISBN: 978-1-4503-1094-9. DOI: 10.1145/2362536.2362544.

[14] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. "The Evolution of System-Call Monitoring". In: *Proceedings of the 2008 Annual Computer Security Applications Conference*. ACSAC '08. 2008, pages 418–430. ISBN: 978-0-7695-3447-3. DOI: 10.1109/ACSAC.2008.54.

[15] *Frama-C: A framework for static analysis of C programs*. URL: http://frama-c.cea.fr/ (visited on 08/01/2012).

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

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

[18] 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.

[19] H. Hartig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. "The Nizza secure-system architecture". In: *Collaborative Computing: Networking, Applications and Worksharing, 2005 International Conference on*. 2005, 10 pp. DOI: 10.1109/COLCOM.2005.1651218.

[20] M. Howard, J. Pincus, and J. Wing. "Measuring Relative Attack Surfaces". In: *Computer Security in the 21st Century* (2005), pages 109–137.

[21] A. Jaquith. *Security Metrics: Replacing Fear, Uncertainty, and Doubt*. 2007.

[22] 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). 2009, pages 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596.

[23] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. "Detecting and countering system intrusions using software wrappers". In: *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*. SSYM'00. 2000, pages 11–11.

[24] 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). 2011, 6:1–6:6. ISBN: 978-1-4503-0613-3. DOI: 10.1145/1972551.1972557.

[25] 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.

[26] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. "Recovery domains: an organizing principle for recoverable operating systems". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*. (Washington, DC, USA). 2009, pages 49–60. ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508251.

[27] Jochen Liedtke. "On $\mu$-Kernel Construction". In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM SIGOPS Operating Systems Review. 1995. DOI: 10.1145/224057.224075.

[28] *lxc: Linux Containers*. URL: http://lxc.sourceforge.net (visited on 08/01/2012).

[29] 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)*. 2010, pages 11–11.

[30] P.K. Manadhata and J.M. Wing. "An Attack Surface Metric". In: *Software Engineering, IEEE Transactions on* 37.3 (2011), pages 371 –386. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.60.

[31] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. "Software fault isolation with API integrity and multi-principal modules". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. (Cascais, Portugal). 2011, pages 115–128. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043568.

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

[33] T.J. McCabe. "A Complexity Measure". In: *Software Engineering, IEEE Transactions on* SE-2.4 (1976), pages 308 –320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[34] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. "TrustVisor: Efficient TCB Reduction and Attestation". In: *Security and Privacy (SP), 2010 IEEE Symposium on*. 2010, pages 143 –158. DOI: 10.1109/SP.2010.17.

[35] 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.

[36] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. "Improving Xen security through disaggregation". In: *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. VEE '08. 2008, pages 151–160. ISBN: 978-1-59593-796-4. DOI: 10.1145/1346256.1346278.

[37] Sarah Nadi and Richard C. Holt. "Mining Kbuild to Detect Variability Anomalies in Linux". In: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*. (Szeged, Hungary). 2012. ISBN: 978-1-4673-0984-4. DOI: 10.1109/CSMR.2012.21.

[38] *ncc: The new generation C compiler*. URL: http://students.ceid.upatras.gr/~sxanth/ncc/ (visited on 08/01/2012).

[39] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. "Faults in Linux: Ten years later". In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. 2011, pages 305–318. DOI: 10.1145/1950365.1950401.

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

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

[42] D. Rosenberg. *CVE-2010-3904 exploit*. URL: www.vsecurity.com/download/tools/linux-rds-exploit.c (visited on 08/01/2012).

[43] B. Schneier. "Attack trees". In: *Dr. Dobb's journal* 12 (1999).

[44] Steven She and Thorsten Berger. *Formal Semantics of the Kconfig Language*. Technical Note. University of Waterloo, 2010.

[45] Yonghee Shin and Laurie Williams. "Is complexity really the enemy of software security?" In: *Proceedings of the 4th ACM workshop on Quality of protection*. QoP '08. 2008, pages 47–50. ISBN: 978-1-60558-321-1. DOI: 10.1145/1456362.1456372.

[46] 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). 2010, pages 33–42. ISBN: 978-1-4503-0154-1. DOI: 10.1145/1868294.1868300.

[47] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. "Reducing TCB complexity for security-sensitive applications: three case studies". In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*. (Leuven, Belgium). 2006, pages 161–174. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217951.

[48] Brad Spengler and PaX team. *grsecurity kernel patches*. URL: www.grsecurity.net (visited on 08/01/2012).

[49] S. Stolfo, S.M. Bellovin, and D. Evans. "Measuring Security". In: *Security Privacy, IEEE* 9.3 (2011), pages 60 –65. ISSN: 1540-7993. DOI: 10.1109/MSP.2011.56.

[50] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. "Nooks: an architecture for reliable device drivers". In: *Proceedings of the 9th ACM SIGOPS European Workshop "Beyond the PC: New Challenges for the Operating System"*. (Saint-Emilion, France). 2002, pages 102–107. DOI: 10.1145/1133373.1133393.

[51] Reinhard Tartler, Anil Kurmus, Andreas Ruprecht, Bernhard Heinloth, Valentin Rothberg, Daniela Dorneanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability". In: *Proceedings of the 8th Workshop on Hot Topics in System Dependability (HotDep '12)*. 2012.

[52] 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). 2011, pages 47–60. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966451.

[53] 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). 2002, pages 604–617.

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

[55] 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*. 2010, pages 51–56.

# Appendix

The following is a proof of Proposition 1.

*Proof* (AS1 and AS2 are attack surface metrics). $AS1_\mu$ satisfies Definition 5 through Lemma 1, as adding new functions to the sum results in a larger attack surface measurement (since $\mu$ has non-negative values).

For $AS2_\mu$, the non-negativity is a known result of algebraic graph theory [6]: the Laplacian matrix of a simple graph is symmetric real and all eigenvalues are non-negative, hence, the quadratic form associated with the Laplacian ($x \mapsto x^T L(G)x$) can only take non-negative values.

Before proving that $AS2_\mu$ satisfies the second property in Lemma 1, we explicit the rationale behind chosing this metric. The metric contains a quadratic term that accounts for the relative "complexity" of a function in comparison to its callers and callees: if a function is calling (or is called by) a more complex function, its relative contribution to the attack surface should increase and vice versa. For instance, this can be written for a function $n$, called by functions $m$, $m'$ and calling function $m''$, $\kappa(n)$ denoting the relative complexity of function $n$:

$$\kappa(n) = \mu(n)\left[(\mu(n)-\mu(m))+(\mu(n)-\mu(m'))+(\mu(n)-\mu(m''))\right]$$

Generalizing to any function:

$$\kappa(n) = \mu(n)\left[deg(n)\mu(n) - \sum_{(i,n)\in\widetilde{C_{AS}}} \mu(i)\right]$$

Which, after summing over all functions, corresponds to ${\mu_{AS}}^T L(\widetilde{G_{AS}})\mu_{AS}$.

Let's now prove that adding a new node to an existing graph can only increase this quadratic term. Without loss of generality, we assume we starting with function set $F = [\![1 \dots N-1]\!]$ and affect $N$ to the newly added function. This function is either called or is calling $m$ functions in $I \subseteq F$ with $deg(N) = m < N$. We denote by $\kappa$ the old relative complexity and $\kappa'$ the new (after the addition of $N$ to the graph), and $deg$ corresponds as well to the old degree of a node, unless it is $deg(N)$. Then:

$$\forall i \in F, \kappa'(i) - \kappa(i) = \mu(i)\left[\mu(i) - \mu(N)\right]$$

Therefore:

$$\begin{aligned}
\kappa(N) + \sum_{i\in F}\left(\kappa'(i) - \kappa(i)\right) &= \mu(N)\left[deg(N)\mu(N) - \sum_{i\in I}\mu(i)\right] \\
&\quad + \sum_{i\in I}\mu(i)\left[\mu(i) - \mu(N)\right] \\
&= \sum_{i\in I}\mu(i)^2 + m\mu(N)^2 - 2\mu(N)\sum_{i\in I}\mu(i) \\
&= \sum_{i\in I}\left(\mu(i) - \mu(N)\right)^2 \geq 0
\end{aligned}$$

Hence, adding new functions can only increase the attack surface measurement $AS2_\mu$. □