FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

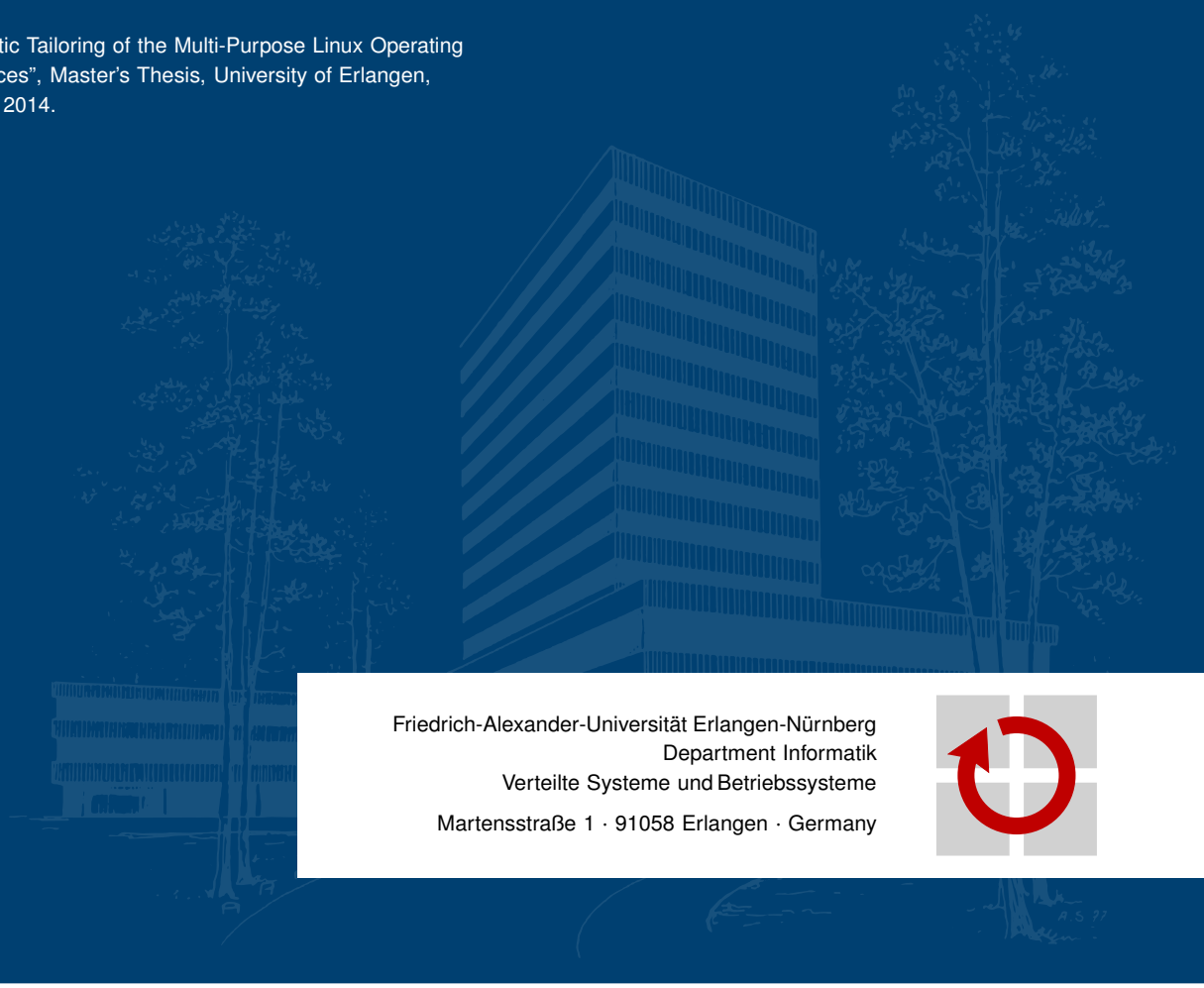
Bernhard Heinloth

Automatic Tailoring of the Multi-Purpose Linux Operating System on Embedded Devices

Masterarbeit im Fach Informatik

September 10, 2014

Please cite as:
Bernhard Heinloth, “Automatic Tailoring of the Multi-Purpose Linux Operating System on Embedded Devices”, Master’s Thesis, University of Erlangen, Dept. of Computer Science, 2014.



Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstraße 1 · 91058 Erlangen · Germany

Automatic Tailoring of the Multi-Purpose Linux Operating System on Embedded Devices

Masterarbeit im Fach Informatik

vorgelegt von

Bernhard Heinloth

angefertigt am

Lehrstuhl für Informatik 4

Verteilte Systeme und Betriebssysteme

Department Informatik

Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuender Hochschullehrer: **Dr.-Ing. habil. Daniel Lohmann**

Beginn der Arbeit: **1. April 2014**

Abgabe der Arbeit: **10. September 2014**

Abstract

Today's system software can typically be configured at compile time using a comfortable feature-based interface to tailor its functionality towards a specific use case. However, with the growing number of features, this manual tailoring process becomes a more and more tedious and difficult task: As a prominent example, the Linux kernel in v3.15 provides nearly 14,000 configuration options to choose from. Even developers of embedded systems refrain from trying to manually build a minimized distinctive kernel configuration for their device – and thereby waste memory for unneeded functionality which increases per-unit costs and restrains the adoption of Linux in cost-sensitive embedded systems.

In this thesis, I present an approach for the automatic use-case specific tailoring of system software for special-purpose embedded systems. By the example of Linux I compare the proposed approach with an existing technique employing virtual machines and evaluate the effectiveness on real hardware by generating tailored kernels for well-known applications of the Raspberry Pi and the Google Nexus 4 smartphone. Compared to the original configurations, my approach leads to memory savings of up to 70 percent and requires only little manual intervention.

Kurzfassung

Moderne Systemsoftware kann üblicherweise zur Übersetzungszeit unter Verwendung einer komfortablen Oberfläche an die Bedürfnisse eines bestimmten Einsatzszenarios angepasst werden. Aufgrund der steigenden Anzahl an konfigurierbaren Merkmalen wird diese manuelle Anpassung jedoch zu einer immer schwierigeren Aufgabe: Ein prominenter Vertreter ist dabei der Linux Kernel, welcher in der Version 3.15 knapp 14 000 wählbare Konfigurationsoptionen bietet. Selbst Entwickler von eingebetteten Systemen vermeiden das manuelle Erstellen einer an das System angepassten, minimalen Konfiguration – und verschwenden dadurch Speicher für nicht benötigte Funktionalität, was die Stückkosten erhöht und damit den Einsatz von Linux in den kostenempfindlichen Bereich der eingebetteten Systeme behindert.

Diese Arbeit präsentiert einen Ansatz für eine automatische, an das Einsatzszenario angepasste Maßschneiderung von Systemsoftware für spezialisierte, eingebettete Anwendungen. Am Beispiel von Linux wird dieser Ansatz auf einer virtuellen Maschine mit bestehender Technik verglichen. Eine Evaluation der Leistungsfähigkeit auf tatsächlicher Hardware erfolgt durch die Benutzung angepasster Kernel in gängigen Einsatzbereichen des Raspberry Pi und des Google Nexus 4 Smartphones. Im Vergleich zur ursprünglichen Konfiguration kann die Dateigröße um mehr als 70 Prozent verringert werden, zugleich ist nur ein geringes manuelles Eingreifen notwendig.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgele gen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.
Alle URL-basierten Quellen wurden, soweit nicht anders angegeben, am 1. September 2014 auf ihre Gültigkeit geprüft.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.
Unless stated otherwise, all URL-based references were checked for validity on September 1st, 2014.

(Bernhard Heinloth)
Erlangen, 10. September 2014

Contents

Chapter 1

Introduction

Most system software can be configured to a broad range of supported hardware architectures and application domains. A usual way is the tailoring to one's need at compile time, the selection is often supported by formal variability models assembling optional features. The most prominent example is the Linux operating-system family, which offers close to 14,000 configurable features across 26 architectures in v3.15 — having an ongoing growth of configurable features by 10–20 percent every year (Figure 1.1)!

This growth appears to be inevitable, as it is mostly caused by advances in hardware: About 88 percent of all features directly deal with low-level hardware support. Especially embedded platforms with many derivatives and short innovation cycles have become a driving force in this process; above all the ARM architecture used in smartphones and tablets as well as developer boards.

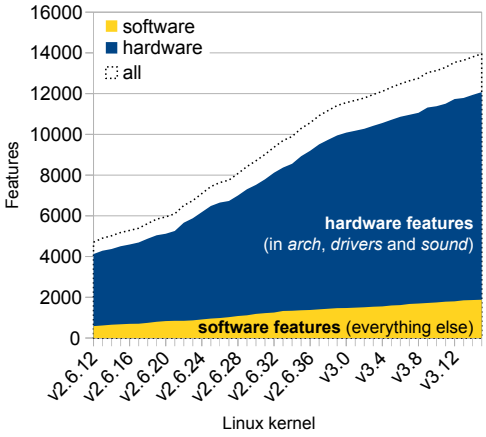


Figure 1.1 – Linux feature growth 2005 – 2014

The downside of these expansions: When configuring a Linux kernel, developers are faced with a way to large number of options. With thousands of features representing possible choices, finding the right set of optional features specifically needed for your system is a hard and time-consuming task — that furthermore requires detailed knowledge about both, Linux and the platform in use. To suit as many customers and their hardware as possible, distributors therefore ship a Linux kernel configuration with most optional features enabled. Instead of per-use-case tailoring, we are practically back to one-size-fits-all solutions.

Although this seems to be a pragmatic approach for workstations with disk sizes of 1 TByte or more and several GByte of RAM, you want your system-software as small as possible if you need it for a special-purpose embedded system. Currently, Linux is already used in smartphones and is the prevailing operating system installed on mini computers like the Raspberry Pi. But there are much more specific use cases for small-scale systems which could be driven by Linux, such as home automation systems or electronic control units used in the automotive industry, where low per-unit costs are a crucial requirement [6].

In the case of Linux, this has led to the development of many special minimized versions, like basicLinux [2], Linux Tiny [34], the commercial Lineo uLinux [32] and Tiny Core Linux[1] [59]. However, these make many assumptions about your system and its usage, trading flexibility for size. And moreover, even on those systems a lot of effort is required by the providing developer to find a valid minimal configuration and keep it up to date for future kernel versions.

I believe it would be easier to take a well maintained standard distribution and automatically derive a configuration specific to the actual needs, once they are known.

Based on the previous work of the CADOS [8] / VAMOS [63] research group, I present a tool-based approach for tailoring Linux on embedded devices by automatically deriving a minimal configuration for a given use case. The resulting configuration can be used by a device manufacturer or embedded systems engineer as a starting point for further refinements. Moreover, this approach is not only limited to Linux but in principal easily applicable for any large-scale system software, since it uses code injections before compile time — in contrast to previous work requiring an extensive tracing environment.

I evaluate the approach on the example of Linux on hardware with two different ARM-based devices (the Raspberry Pi and the Google Nexus 4 smartphone) in four real world scenarios — leading to net memory savings of up to 70 percent compared to the original configurations. Additionally, I establish an emulation framework providing the ability for a detailed comparison between different implementations, supporting future improvements by engaging it as a verification platform. Lastly, this framework can help identifying the limitations of the general approach by instrumenting a virtual machine.

[1] Tiny Core Linux is under active development and supports several platforms including Raspberry Pi.

The remainder of this thesis is structured as follows: In Chapter 2, I present an overview of how variability is implemented in Linux, the concept behind the previous and the new approach as well as the related work on this topic. I depict a detailed description about the implementation in Chapter 3, including the reasons for related decisions made in this project. Subsequently, I demonstrate the application on hardware in various real life use cases with respect to kernel size metrics in Chapter 4 and an emulation framework with the ability for detailed comparison in Chapter 5. Afterwards I discuss these results as well as the limitations of the approaches in Chapter 6.

Parts of this work have also been published with Andreas Ruprecht and supervised by Daniel Lohmann as "Automatic Feature Selection in Large-Scale System-Software Product Lines" [45] at the 13th International Conference on Generative Programming: Concepts & Experiences (GPCE'14). While Andreas mainly focused on the textual part, I was responsible for the development and the evaluation[2]. Substantive decision concerning the approach were made together in the VAMOS research group.

[2]Since both Andreas and I contributed almost equal fundamental parts, we simply decided the order of authors by the roll of dice — our normal procedure.

Chapter 2

Background and Scoping

In the following section, I first give an idea how static variability is implemented in Linux, that is, how configurable features and their constraints determine the resulting binary code. To retrieve a tailored configuration, a reversed mapping of used binary code to source code is necessary. An automated process (utilizing FTRACE) demonstrates the basic approach in the consecutive section. Faced with the limitations of previous work, I show the need of revising the approach to make it applicable on embedded devices. Therefore I declare my objectives and outline the suggested improvements based on code injection. This leads to various challenges, which I present in the subsequent section before finally examining related work on this topic.

2.1 Variability in Linux

Configurability in Linux is basically specified using the KCONFIG language [26]. In KCONFIG, a kernel developer can describe a configuration option — denoted as feature — which can be selected when specifying features desired in the kernel. Additionally, constraints and interdependencies between configuration options can be specified. For example, for a USB audio device it is necessary to build general USB support into the kernel; the developer would hence describe the configuration option for the device as dependent on USB support. Due to these obvious feature dependencies, the KCONFIG features are organized in a tree-like structure. But the activation of a feature in one part of this tree does not only enable all parent feature nodes down to its root — in addition, it can (and often does [4]) trigger the selection or deselection of features in other branches of the tree[3], depending on the preconditions described by the developer.

[3] Having these horizontal dependencies the structure cannot always be modelled as a tree but instead as a directed acyclic graph — although theoretically the KCONFIG language allows circular dependencies.

Therefore the feature models become quite complex — too complex for a pure manual configuration of the Linux kernel. In practice, the user first selects the hardware platform via the `ARCH` environment variable and can then choose from all KCONFIG features available on this platform with a graphical or text-based configuration tool which ensures that the resulting configuration is valid. Several of these features cannot only be enabled or disabled, but marked as "Module"[4], enabling the kernel to load these features on demand.

All the options selected and deselected are gathered by KCONFIG in a single kernel configuration file called `.config` inside the kernel source directory.

The configuration is then interpreted by the build system to implement coarse-grained variability. Depending on the selected features, KBUILD determines which of the more than 38,600 files[5] need to be compiled and linked to include the selected features. In Linux this is the dominant mechanism to implement variability: In version 3.15 almost three-quarter of all KCONFIG features are used to guide the build system in this way[6].

On the thereby selected source files, the C preprocessor (CPP) is used to implement fine-grained variability via conditional compilation (`#ifdef` blocks). In Linux 46 percent of all KCONFIG features are interpreted in this step to select from a total of more than a hundred thousand conditional blocks.

Lastly, MAKE is used to set the correct compiler options, determine the binding units and generate the Linux kernel image and any corresponding loadable kernel modules as specified by the KCONFIG selection.

2.2 Basic Concept for Automatic Tailoring

In order to obtain a Linux configuration tailored to a specific scenario, I need a strategy to reverse this process, that is, to find exactly those features that select (only) the required parts of the code base.

The idea to obtain them is to run a use-case–specific workload and concurrently observe which parts of the binary code are executed[7]. You then need to determine the reverse mapping (via conditional blocks, build rules, and feature model) to those features that have to be selected in order to have these specific code parts in the resulting binary.

[4] As long the kernel supports loadable modules, triggered by the option `CONFIG_MODULES`.

[5] Kernel source files (assembly, C code or header) without helper tools/scripts or samples.

[6] Dietrich et al. [17] published similar results for the Linux kernel version 3.1.

[7] A very rough method is already shipped with Linux: `make localmodconfig` builds a configuration based on the currently loaded kernel modules. This might be a handy solution for workstation users, but it is far too imprecise for professional application in embedded systems, since only complete modules and all statically enabled features are taken into account. Further information can be taken from the official announcement at `http://article.gmane.org/gmane.linux.kbuild.devel/3750`.

2.3 Previous Work

In an earlier approach described in a workshop paper [55], the VAMOS research group already successfully leveraged the FTRACE infrastructure [20] to automatically tailor Linux kernels for web server and workstation use.

FTRACE is a frame work built into the Linux kernel which can be used to gain insight on the control flow within the kernel. The activation of FTRACE on a prepared kernel provides a profiling interface to the user making it possible to track which kernel functions are executed during run-time.

2.3.1 Procedure

The traditional tailoring approach consists of four basic steps, which are also depicted in Figure 2.1:

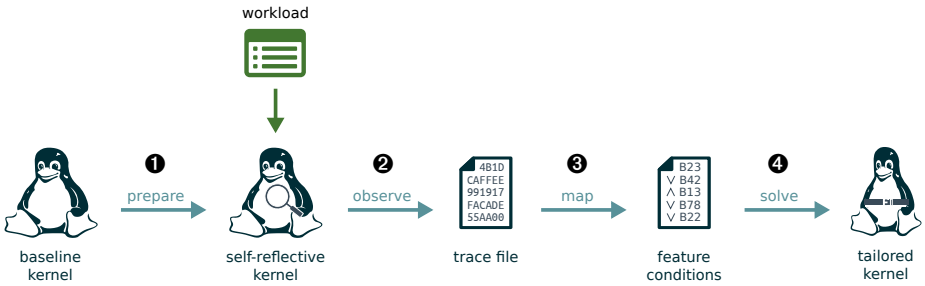


Figure 2.1 – Overview of the kernel tailoring approach

❶ **Preparation:** The corresponding KCONFIG options[8] instruct the compiler to use the profiling functionality: It inserts a call to a specific `mcount` function at the beginning of each translated function which is itself implemented by the FTRACE infrastructure. Activated by additional KCONFIG options[9], the compiler includes debug information in the binary offering the ability to draw conclusions from the original source code position.

❷ **Observation:** After booting the system with the prepared Linux kernel, a target workload — adapted to the use case — will be executed on the system. This will lead to additional functionality being triggered in the kernel.

The VAMOS tailor tool collects the data from the kernel by reading and parsing the output pipe of FTRACE while running the workload, as FTRACE can only buffer a limited amount of information. The addresses of executed functions are then written into a separate output file.

[8] Mainly the tracing infrastructure enabled by the feature `CONFIG_TRACER`.
[9] Represented by the feature `CONFIG_DEBUG`.

After the workload has been run, you save the output file for further processing as described by the following steps.

❸ **Source code mapping:** In this step the information obtained from step ❷ is processed. The VAMOS tools use the kernel's debug information to resolve the addresses obtained from the output file to the corresponding locations in the source code.

You now have a list of file names and line numbers of code that have been executed in the measured scenario. For every item in this list, the preconditions described by the conditional blocks around the code have to contain dependencies described by KCONFIG. Tools described in previous work [49, 56, 16] are able to determine the preconditions described in KCONFIG and provide an option to look up the preconditions for a given line.

A description of the complete conditions for the whole scenario observed is obtained by conjugating all individual conditions into a propositional formula.

❹ **Solving:** To derive a valid configuration from this list of features and preconditions generated by step ❸, a (boolean) satisfiability problem (SAT) solver is employed. The resulting assignment of variables represents the selection or deselection of configuration options for the kernel.

As the configuration system itself might enforce additional constraints not covered by the extracted dependencies, this partial configuration is lastly expanded by the KCONFIG system, generating a fully valid Linux kernel configuration. This configuration can either be used to directly compile a tailored Linux kernel or be used as the base for further refinement by a developer.

2.3.2 Limitations

Utilizing FTRACE to observe which parts of the code were actually executed worked well on previously tailored x86 machines. However, I discovered it is not generally applicable for the generation of small kernels on weaker ARM systems, as it induces high overhead during the observation phase. For example, FTRACE records additional information about latency and execution time (and presents the data in a comparably verbose way), therefore taking up a lot of computation time itself. This circumstance lead me to the next problem: The usage of FTRACE guided by UNDERTAKER tools in user space causes not only a high resource consumption (both in memory and computing power) but also many side effects[10]. Of course, the approach is only

[10] You are not only able to discover these side effects in the memory management and process scheduler but, for example, in the debug- and root file system access, too. The execution of payload will suffer from the limited memory and computing time and therefore potentially react in particular way.

applicable on specific traceable kernels compiled with both debug information and activated tracing/profiling environment.

Since FTRACE uses the profiling environment triggered only on function calls the granularity of the traditional approach is basically limited to function level — the fine-grained variability implemented by conditional compilation is not taken into account.

Last but not least, the portability of this approach to further operating systems is limited as it requires an extensive tracing infrastructure — even Linux itself does not provide tracing support for every supported architecture: For example, the Motorola 68000 series is not supported. Therefore, I assume a generalization of this existing approach to other operating systems and software product lines is quite difficult to manage.

2.4 Goals of this Thesis

To make an automated tailoring applicable for resource constrained embedded devices, it is necessary to reduce the overhead by avoiding the extensive FTRACE environment. Therefore, I have to develop a less-invasive code-point recording method regarding computing time and memory consumption. By keeping the interaction in userspace as small as possible, I will be able to reduce undesirable side effects.

Previous work only verified a small number of manually guided tailoring examples since this process can be quite time consuming[11]. This might be necessary to demonstrate the practical relevance, but it is not suitable for a detailed comparison of different approaches: Because of the manual intervention, the interactions involved in processing slightly varies and produce imprecise results. Due to this limitation, I prefer a methodical evaluation: Having an identical environment and automatically triggered interactions I expect better comparable results. To achieve this, I suggest building a framework engaging a virtual machine with the possibility to take all approaches into account. It should be embedded in the CADOS infrastructure for regression testing and supporting future developments.

Since its development is (in principal) straight forward, I will develop this framework after successful investigation of the new code recording approach.

2.4.1 Suggested Approach

For any method based on tracking kernel activity the principle remains the same — it is necessary to indicate executed code which depends on kernel configuration

[11] It usually takes more than five hours from the generation of the traceable kernel to a ready-to-use tailored kernel — without calibration of whitelist files, of course!

choices. To meet the disadvantages of the previous FTRACE based version, I suggest a lean and simple approach: At the beginning of each function a specific statement logs its execution. This level will suffice for most features in the Linux kernel because the gross of them are a rough granularity: They only affect whole functions or files [17]. At least in theory the new approach offers possibilities to increase the detection: Because of the fact that configurability inside source code is mostly done by CPP controlled blocks, it is possible to cover almost the complete configurability by injecting into these blocks.

It seems that the easiest way to fulfil these requirements is a source code modification before compile time — due to the fact that compilers perform CPP and code analysis in different steps. For this purpose I inject specific commands (as CPP macro) to each of these major feature-dependent code block (demonstrated in Listing 2.1). During the execution, an injected command switches a unique boolean value marking the code point as 'executed'. For the sake of simplicity, each boolean value is a particular bit in an exportable global memory map. Since the job is basically just flipping bits, for better understanding this new concept is referred to as FLIPPER for the remainder of this work.

```
1  #include <linux/do_sth.h>
   +#include <linux/macro.h>
2
3  void baz(){
   + INSERT_MACRO1_HERE()
4    do_sth(42);
5  }
6
7  #ifdef CONFIG_FOO
8  int foo(int i){
   + INSERT_MACRO2_HERE()
9    return do_sth(23*i);
10 }
11 #endif
12
13 int bar(int x){
   + INSERT_MACRO3_HERE()
14   int i=0;
15 #ifdef CONFIG_FOO
   +    INSERT_MACRO4_HERE();
16     i=foo(x);
17 #else
   +    INSERT_MACRO5_HERE();
18     i=x;
19 #endif
20   return i;
21 }
```

Listing 2.1 – Example of code injection concept

2.4.2 Procedure

The procedure will be similar to the one described in previous work, Figure 2.2 illustrates the differences:

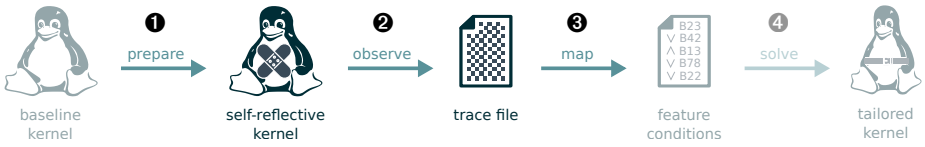


Figure 2.2 – Modified steps in the newly suggested kernel tailoring approach

❶ Preparation: FLIPPER analyses and patches the Linux source code before the compilation starts. Similar to `mcount` in the FTRACE approach it places an instruction at the beginning of each function. Its task simply consists of switching a specified bit in a global bitmap. Moreover, this instruction can be placed in every feature-dependent block (denoted by the `#ifdef` directive).

❷ Observation: Using the system in a predefined scenario ensures all required functionality of the kernel will be called. In doing so the newly injected commands from step ❶ are executed, allowing us to draw conclusions from the code actually used in the workload. Unlike the traditional approach, you only have to read the bitmap from the system once the target workload has finished running, as the bits have gradually been set during execution.

❸ Source code mapping: Afterwards you have to evaluate the output bitmap file from step ❷. Whenever a bit is set, you collect the associated entry from the mapping file generated during step ❶.

Since you have now a list of source code positions, from this point the process is identical to the FTRACE based approach described in previous work.

2.4.3 Challenges

In order to come up with a thorough solution for deeply-embedded systems, the approach described has to face some challenges:

Invasiveness Collecting the information about which parts of the code have been executed must only minimally affect the observed system's behavior. While FTRACE was successfully used to tailoring Linux on a high-end x86-64 server machine, it proves to be too complex for its application in a weaker system. Trying to use FTRACE on resource-constraint devices results in altered timing behaviour and important information about executed functions being dropped from the output buffer, which are then not being accounted for in the resulting configuration.

Accuracy At the same time, it is important to gather as much information as possible to correctly model the configuration requirements for a given scenario. As described above, FTRACE fails to accurately collect all data due to unneeded overhead. Especially during the early boot phase, which triggers a lot of functionality, function calls representing critical features can easily be missed.

Completeness of the recordings By design, my approach can only take information into account which has been triggered during the observation phase. This, however, should not cause the tailored system to fail if additional functionality related to the triggered functionality — for example, error handling in a driver, when no error occurred while running the target workload — is needed during later productive use.

Untraceable features Moreover, some configuration options like errata specific to a certain processor or compiler flags which do not have an immediate representation in the control flow, might not even be detectable at all. This requires external knowledge to be taken care of while deriving a solution. This particularly applies to KCONFIG features of string or numeric type (for example the kernel command line or section offsets), where an automated solving approach cannot provide any choice.

Alternatives Some KCONFIG features present a set of alternatives to the user (e.g., the choice of a scheduling strategy). From these, the SAT solver will simply choose one, as there are no further constraints to observe. Additionally, the default choice provided by the distributor might not fit the systems actual needs. Thus, the developer needs to be able to specify previously known selections to integrate his domain knowledge into the tailored kernel.

2.5 Related Work

In earlier work [55, 28], the VAMOS research group has been able to show the general feasibility of tailoring a Linux kernel to a specific use case, observing improvements in binary size and security. As already discussed, however, the approach presented there needs comparably strong hardware to cope with the amount of data generated during the observation phase, rendering it useless for application in embedded systems.

There are a number of other researchers working in the field of specializing configurable systems, whose findings I will briefly outline:

As an example, Lee et al. [29] use a graph-based approach to identify the specific needs of an application and the underlying Linux operating system. They subse-

quently remove all code not required by the target application (e.g. unnecessary exception handlers and system calls) from the source code.

Chanet et al. [9] also propose the analysis of a control-flow graph of both the applications and the Linux kernel. Instead of patching the source code however, they use link-time binary rewriting to eliminate unused code from the resulting compiled kernel.

For embedded devices based on Linux and L4 Bertran et al. [5] suggest a "global control flow graph": Their approach eliminates dead code in binaries emanating from entry points defined by the application binary interface.

A shared drawback of these approaches, however, is that they do not make use of any configurability options already provided by the kernel, which could eliminate code as well. Moreover, by patching information out of the binary they are prone to leaving "loose ends" inside the kernel. My approach in contrast is assisted by the configuration system itself. This ensures a valid Linux kernel configuration is derived and used for compiling the tailored kernel.

An approach taking configurability into account when deriving a tailored software system has been presented by Schirmeier and Spinczyk [46]. Again, static analysis is used to determine relevant parts in the code, the authors however only tested their work on a much smaller and less complex application with only 15 configurable features, already leading to a graph consisting of approximately 600 nodes.

In contrast, Siegmund et al. [47] use interacting configurable features to predict non-functional properties like performance from a given configuration, and also developed a method to automatically derive an optimized software variant [48]. Perhaps it would be interesting to combine these results with my tailoring approach; for example, the generation of a tailored configuration could not only consider selecting as few features as possible, but rather select features optimal for non-functional properties deemed important for the target use case, e.g. power consumption in an automotive scenario.

On the other hand, my results could be used to extend their work onto the Linux kernel. While this has not been feasible to date due to the massive amount of KCONFIG features in Linux, the authors could reduce the problem to the features (and their possible alternatives) identified by the tailoring approach.

To integrate preferences of the user while optimizing a configuration for non-functional properties, Soltani et al. [50] model the selection of features as a Hierarchy Task Network (HTN) planning process. Due to the run-time of their approach already rising strongly when applied to a random model consisting of only 200 features, its adaption to a real-world large-scale system could prove to be very difficult, if not impossible.

Guo et al. [23] present a genetic algorithm to find an optimal feature selection incorporating resource constraints in a software product line, which also performs well for a randomly generated model consisting of 10,000 features. The generated configuration, however, is not use-case specific: The optimization is performed using cost vectors associated with every feature (i.e., CPU or memory consumption) rather than considering specific functionality requirements deduced from actual system use.

2.6 Summary

Tailoring an extensive Linux configuration of a standard distribution down to ones need can be achieved by recording the actual executed code while running an exemplary work load. The previous approach presented by the VAMOS research group is based on the Linux tracing infrastructure FTRACE, which is either too resource intensive or even not available on some architectures. As a solution, I suggest injecting special instructions into the source code before compile-time to record the executed code points. I will evaluate the new approach both on real hardware to demonstrate its application and — for better comparison with the previous approach — in an emulation framework.

In contrast to the approaches presented by other researchers, this work focuses on a transferable solution which can support system engineers since it makes use of KCONFIG (the Linux configuration system) and is applicable for daily use in real-world scenarios.

Chapter 3

Design and Implementation

With respect to the difficulties on manipulating C source code with CPP macros (briefly described in the following section), I decided to develop a rough proof-of-concept prototype from scratch to gain experience about the new code-point recording approach. For the final version I consider several third-party tools focusing on source-code transformation and take the experience made with the previous prototype into account.

3.1 Problems on Code Manipulation

While the transformation of source code to an abstract syntax tree (AST) is a well explored challenge, the source-to-source transformation of C code including CPP macros remains a hard problem [51]: Due to historical reasons, the macro language is not part of the AST but a line based preprocessor task before the actual C compilation.

> The preprocessor was originally considered an optional adjunct to the language itself. [...] This attitude persisted, and explains both the incomplete integration of the syntax of the preprocessor with the rest of the language and the imprecision of its description in early reference manuals. *(Dennis Ritchie, inventor of* C *[44])*

Compilers process C code (to an AST) after successful interpretation of all CPP macros[12] — while output as pure C code is still possible[13], the full procedure regarding macros is non-reversible. The preprocessor has the ability to include files (`#include`), expanding macros and controlling compilation of code segments

[12] According to the standard [24], the evaluation of the preprocessor directives and macros (and deletion of all remaining ones) is the fourth step of the translation phase, while the AST is generated after finishing all eight preprocessor steps described in the standard.

[13] This is often used to format/beautify code (also called “pretty printing”).

(`#ifdef`, which is mainly responsible for the fine-grained variability in Linux). In particular, the use of the compilation control is not limited to complete C statements but can slice expressions in several parts — therefore it is not possible to consider them in the C-AST. However, only the AST allows an automatic source modification (without accidentally changing semantics).

Notwithstanding, several researchers tackled this problem and developed C source manipulation tools like PUMA with the ability to preserve CPP macros. But using them is not trivial at all, because of either extensive application programming interface (API) or lacking documentation. Hence they are initially not taken into account at the development of the prototype, while — after successful investigation of said prototype — a full featured version regarding such C source transforming projects is developed.

3.2 Prototype

Since wide parts of the Linux kernel source really comply the coding guidelines [33], it seems to be sufficient to use regular expression (RegExp) to analyse the structure of the C source. Therefore, I choose PERL as the implementation language for the prototype because of its comprehensive RegExp support including recursive patterns[14] (which are necessary for parsing unlimited nested parentheses). Unlike compilers, my tool processes lexical and syntax analysis together in one step. Using fixed-point iteration algorithms, irrelevant lines are reduced according to the C grammar [27, p. 193ff], leaving just a basic structure of the source. Of course, this tentative implementation is not focused on incorporating the whole grammar — for my prototype I concentrate on its main parts used in the Linux source, always bearing possible flaws in accuracy in mind.

To prevent extensive processing of all Linux source files each time deploying the approach to the kernel with its 38,600 source files, I prefer patch file output (in "unified diff format" [15]) as an intermediate step instead of direct file modification: Only a single analysis run is required for each kernel version and it takes only seconds to apply these preprocessed changes to the Linux source.

For better comprehension in the later parts I named the prototype DURDEN — in contrast to the final tool named FLIPPER (according to the concept).

[14] "Regular expressions" within the meaning of formal language theory are only able to match patterns in regular languages (type-3 grammars in Chomsky hierarchy [10]). But since most programming languages are context-free (type-2 grammars) — for example, they allow recursive structures — only the expanded PERL-RegExp implementation has the ability to deal with these languages.
Note: Due specific constraints in its standard, C is in fact a context-sensitive language (type-1 grammar) [21] — but this has no influence on the development of the prototype.

3.2.1 Code Injection

Special FLIPPER commands need to be inserted in source code locations indicating a feature-dependent code block.

Since the kernel will execute the injected code many times — thus, having a big influence on overall performance —, it needs to be as slim as possible. Instead of the concrete implementation, my approach introduces a CPP macro to each code point enabling me to choose the concrete implementation afterwards.

Each inserted macro has just a single argument: A unique number as identifier (ID) indexed by the injection routine. This ID allows the identification of the corresponding memory allocation and enables linking executed macros to a concrete source line.

After compilation, every injection enlarges the kernel — depending on the architecture and optimization flag — by a net amount of one (x86 with optimization enabled) up to eight (ARM without optimization) instructions (cf. Table 3.1 for a more detailed comparison).

target architecture	compiler optimization	instructions[15] for performing bit flip	byte set
arm (RISC)	disabled	8: `ldr ldr ldrb orr uxtb ldr ldr strb`	3: `ldr mvn strb`
	enabled	4: `ldr ldr ldrb orr strb`	3: `ldr mvn strb`
x86 (CISC)	disabled	3: `movzbl or movb`	1: `movb`
	enabled	1: `orb`	1: `movb`

Table 3.1 – Comparison of required assembly code instructions for approaches compiled on ARMv6 and AMD64/x86-64 architecture using GCC with optimizer flag `-O2` (or `-O0` in case of disabled optimization)

In practice, the concept of flipping just a single bit suffers from concurrency flaws: On many architectures it is not possible to do an atomic bit flip. But since it is possible to have multiple injected macros executed at the same time on multicore (or during scheduling even on single core) systems race conditions can occur. While a mutual exclusion lock (mutex) for each macro seems to incur too much overhead, I prefer solving concurrency by using the smallest, direct addressable data type[16] for each macro instead of a single bit. On the one hand this increases the size of the map by the number of bits used in this data type[17], but on the other hand no additional expensive locking operations are necessary. This option can be activated by module configuration.

[15] This collection lists only the machine code instructions directly involved in the macro. Additional instructions to recover registers may be inserted depending on the code as well as two additional words required for bitmap memory address computation on arm.

[16] Usually a one byte character.

[17] Obviously eight times for byte-wise access.

3.2.2 Kernel Module

To handle and retrieve the data collected by FLIPPER, I decided to create a kernel module with a userspace interface. Each bit in a global bitmap represents a well defined code point — relocatable by the identifier. On execution, the injected macro propagates the usage of the code point by activating the corresponding bit in a global map. The kernel module deals with exporting this bitmap as a symbol in order to make it accessible from anywhere in the kernel — including loadable kernel module (LKM) — and implements a char device providing the ability to access its data. The bitmap size is hard-coded at compile time; to configure the size of the bitmap, KCONFIG is employed[18].

3.2.3 Function Injection

Injecting a function with the basic source structure is quite easy: The tool performs a macro injection at the first position immediately after any initial variable declarations[19]. Only a single restriction is necessary: You must not apply the macro to source files employed outside the kernel context since you cannot access the global map from the kernel module in these cases (and, moreover, these parts are mostly not relevant for tracing). Basically the Linux kernel has three types of such source files:

- Tools only involved in the kernel compilation process
- User space libraries
- Routines utilized on early boot (such as unpacking the kernel)

The automatic detection of such files is a give-away of the build system: The compilation process will report a missing symbol (referring to the global bitmap). In order to address this issue, I incorporated a blacklist which avoids the parsing of defined files. It turned out that these problematic files are pretty static — once I set up the blacklist (presented in A.3), there was no need to change them later (not even when switching the Linux kernel version).

However, LKMs can be triggered even if there was no actual module interaction since Linux provides initialisation calls: The function referred to by the `module_init()` macro is automatically executed after the module was successfully loaded into memory. To avoid this, I prevent functions denoted by such special macros from being injected[20].

[18] It is absolutely necessary to choose a size at least of the quantity of all injection points — otherwise overflows can occur, which may cause a kernel panic or an undefined behaviour.

[19] Since the kernel compiles with the flag `-Wdeclaration-after-statement`, this is necessary to avoid a flood of warnings at compile time.

[20] However, my detection method is limited: For example, it does not support special handling for helper functions called from the initialisation routine — once helper functions are triggered by the tool, these may mislead the SAT solver to include unused modules. For further information see Section 6.1.

3.2.4 Block Injection

As mentioned earlier, features can sometimes be very fine-grained (although it does not happen often): It is possible that a configuration choice enables just a single source-code line within a function — or even just a part of it — by using CPP macros. To cover all code variability, I implemented a heuristic routine to handle these feature-depended CPP blocks in the Linux source code. The fundamental procedure is similar to functions: My tool has to introduce code at the beginning of each CPP block. While the macro injection of blocks containing only complete simple statements[21] is easy, I was faced with a wide range of more complex applications[22]:

- Compound statements[23] may require extra treatment: A block inside a `switch` `case` statement including multiple cases needs to be injected multiple times — the macro must be placed after each `case` (or the `default`) keyword to guarantee its execution on every condition value (an example is provided in Listing A.2). Cascaded `else if` conditions are handled in a similar way.

- Single statement blocks without curly braces (appearing after compound statements) require special treating, too. I figured out that instead of applying the block indicating braces, the comma operator is the least invasive injection (like shown in Listing A.1). The broad range of possible applications is especially shown in case of using the shorthand `if` syntax, but this comma operator injection seems to be the best practice in complete expressions anyway (similar to Listing A.3): First the corresponding code of the macro is evaluated (while its return value is discarded) and the result of the second (original) statement is used after its evaluation for the further proceeding.

- Conditional blocks can — and will — occur inside expressions. To place my macro, I make use of the comma operator again and construct an identity transformation[24]: The first operand utilizes the activation of the corresponding bit, while the second one is just the constant "0": `( MACRO() , 0 )`.

  The algebraic addition operator (denoted by the plus sign "+") concatenates the new expression to the existing one inside the block — its position naturally depends on the placement of the previous operator (cf. Listings A.4 and A.5). Although this solution works in practice with current Linux kernel versions without restrictions, hypothetically this procedure can change the semantics of the code: Multiple expressions connected by operators with

[21] For example, assignments and function calls.

[22] Liebig, Kästner, and Apel [31] denoted them as "undisciplined annotations", accounting about 4 percent of the CPP usage in the Linux kernel v2.6.28, besides 3 percent blocks which could not be classified.

[23] Like conditions and loops — characteristic trait: It includes additional statements.

[24] An operation which does not change the value.

different precedence are evaluated according to the order of operations — the example Listing 3.1 will accidentally multiply the identifier *b* with the constant "0" and actually erase its value. At first sight, the multiplication operator (denoted by the token "∗") in conjunction with the constant "1" seems to be the better identity relation referring to its operator precedence. Since it is not possible to use the multiplication operator on pointer type expressions (this will cause a compile time error), I refused this concept as the tool will not perform type checking.

```
1 a = b *
2 #ifdef CONFIG_FEATURE
3 +( ( SET_DURDEN_BIT(23) ) , 0 ) +
4   c +
5 #endif
6   d;
```

Listing 3.1 – Pathological example presenting limitations of the approach

Nevertheless, there still remain a few special constructs, which I cannot correctly cover by my common heuristics mentioned above. Their detection is not trivial: A flawed injection can either cause a compile time error (in case of invalid syntax), lead into run-time errors or — even harder to recognize — incorrect behaviour without visible errors by changing the semantics. An exemplary code snipped from Linux kernel v3.6 is presented in Listing 3.2. However this particular code was revised in later versions[25], I could find similar examples in each kernel version, especially located in the drivers section.

```
686   entry.saddr =
687 #if IS_ENABLED(CONFIG_IPV6)
688     (entry.family == AF_INET6) ?
689     inet6_rsk(req)->loc_addr.s6_addr32 :
690 #endif
691     &ireq->loc_addr;
```

Listing 3.2 – Example of Conditional block inside expression with prefix operator (Linux v3.6 source file `net/ipv4/inet_diag.c`)

For testing purposes, I overcome this problem by blacklisting such files as a short term solution; I was able to figure out about twenty files with incorrect injections (files listed in A.3).

[25] But this code was part of the kernel for eight years: It was introduced in kernel version 2.6.10 (December 2004) in *net/ipv4/tcp_diag.c*!

3.3 Final version

After extensive testing of the prototype DURDEN, the experiences are taken into account to develop the final version. Due to the problems in modifying CPP macros (see 3.2.4), I decided to abandon the concept of CPP block injections: Comparison of various tests revealed no notable difference to a configuration file obtained with only function entries patched[26] — actually I was not even able to create a single Linux test case which perceptibly benefits from the additional block injections.

The final version, called FLIPPER (corresponding to the approach), only adopts the principle of injecting whole functions once from the prototype (described in 3.2.3), which allows me to record just the execution of every function. This decision ensures a long-term application of the approach without the need to care for a new file blacklist in each new version.

In contrast to the prototype, third party tools are involved to guarantee a semantically correct macro injection. I choose PUMA, LLVM/CLANG and COCCINELLE for further investigations.

3.3.1 PUMA

The PURE MANIPULATOR (PUMA) used in ASPECTC++ [1] (and UNDERTAKER) has the ability to handle CPP code while manipulating C sources. It is integrated in the UNDERTAKER tool, and the VAMOS team is familiar with its interface. However, its development has almost stopped and the ASPECTC++ team announced the integration of LLVM/CLANG instead of PUMA for the upcoming version 2.0[27], which could indicate the definite end of its development. Under these circumstances I considered an implementation based on this library disadvantageous and discontinued work.

3.3.2 LLVM/Clang

Due to historical reasons, Linux is closely connected to the GNU COMPILER COLLECTION [22] (GCC), although the young project LOW LEVEL VIRTUAL MACHINE [60] (LLVM) with its C language frontend CLANG [12] is becoming more and more popular in the last few years. Recent projects[28] try to port the kernel to the LLVM/CLANG compiler, though these projects have unsolved issues and are not in a stable state yet. One advantage of CLANG is the well-documented and simply modifiable code base[29] compared to GCC.

[26] A more detailed description can be found in Section 6.3.

[27] As announced by a ASPECTC++ project member in a personal conversation.

[28] Like the popular LLVMLINUX project [35].

[29] This advantages are claimed by the developers on their official comparison site: `http://clang.llvm.org/comparison.html`.

Providing tools for modifying the AST and rewriting source makes CLANG a perfect starting point for a clean implementation. However, first attempts of the source-to-source transformation pointed out that the AST is generated after handling the CPP macros with the available context information: CPP blocks enabled by KCONFIG options are ignored in the AST. At the code rewrite step, these ignored parts are pasted untouched to their origin code points.

```
1  int foo(){
   // Flipper code would be inserted here
7    return 23;
8  }
9
10 #ifdef CONFIG_BAR
11 int bar(){
12   return 42;
13 }
14 #endif
```

Listing 3.3 – Code injection by CLANGS source rewrite engine ignores KCONFIG enabled conditional blocks

Since the KCONFIG macros are evaluated at kernel compilation time, they are not available during preparation. Combining the steps code analysis, patching and compiling could be a solution, but I rejected this approach owing to its need for extensive modifications and the open issues in the LLVM kernel projects mentioned earlier.

3.3.3 Coccinelle

In contrast to the traditional patch format, the semantic patch language (SmPL) suggested by *Laboratoire d'Informatique de Paris 6* is independent from line numbers: Source-code lines are referenced by their semantic structure. Moreover, a single patch is not limited to a single file but can modify thousands of files without knowing them at creation. Since the beginning of development, Padioleau, Lawall, and Muller [39] focused on Linux as primary target with the purpose to get a grip on the collateral evolution problem (and it already made its way into the Linux kernel source).

Even though their open-source application COCCINELLE is the only available tool able to interpret this language, SmPL seems to fulfil my requirements: I decided to base the final version upon COCCINELLE/SmPL.

Although the language was not originally created for targeting use cases like described in this thesis, the integrated PYTHON support enables a wide range of functionality beyond source code modification: The generation of the mapping file

(consisting of bit field number, file name and line number for each entry) as well as the blacklisting engine make use of PYTHON. Hence it is sufficient to run a single COCCINELLE instance with the full Linux directory as argument instead of employing an additional script guiding through the files.

Albeit COCCINELLE can modify the Linux source directly, I suggest generating a traditional patch file for the whole source as common usage. My experiences with the prototype exposed this progression as best practice: A portable patch can be applied to an appropriate kernel version in seconds — and be revoked easily.

Drawbacks

Although the code parsing of the new approach is based on — theoretically — the full C language grammar (instead of the partial implementation in the prototype parser), and I was able to significantly improve the code readability of my tool at the same time, there are still some noticeable drawbacks left:

- Since the source code injection of a file with the COCCINELLE approach requires the generation of a complete AST, it is a more time consuming procedure than the prototype. For instance, the final FLIPPER version processes a Linux kernel in about 90 minutes while the prototype DURDEN needs less than five minutes on the same software/hardware configuration.
- Unlike the prototype, it is not able to patch null functions: Due to limitations of SmPL in the latest version, it is not possible to address empty function bodies for inserting the macro. Although this affects about 1,500 functions, I figured out that the missing lines have no measurable impact.
- Similarly to above, it turned out that an insertion of `#include` directives using this tool is not as simple as you could expect: Since SmPL needs semantic context to attach new code, you cannot address the top of files without other CPP directives in it. I handle this special case by inserting a directive in front of each patched function — with include guards[30] solving the possible occurrence of multiple insertion.
- Lastly, COCCINELLE mismatches functions defined in multiline CPP macros as normal functions (like A.6). Performing an injection to files with such definitions will invalidate the syntax and lead to a Linux compile error. Since the interpretation of such code in the AST is an incorrect behaviour and just about a dozen files in the current kernel are affected, my preliminary solution is quite simple: I add the files to the blacklist until the developers of COCCINELLE fix the problem.

[30] A common way to prevent multiple processing, described in `https://gcc.gnu.org/onlinedocs/cpp/Once-Only-Headers.html`.

3.4 Summary

Automatically injecting code in all Linux C source files without changing (or breaking) semantics is a difficult task. I developed a prototype from scratch with not only the ability to extend all functions in Linux by a recording macro, but also (correctly) inserting such macros into most of the conditional blocks present in the source. Only a few occurrences could not be assigned to the right working set, thus leading to incorrect behaviour as long as you do not manually exclude the respective source files. However, it turned out that I get the same result, whether I just tracked function calls or in addition enabled the extensive block recording — but having a higher overhead and error rate in the last case.

Therefore, I rejected the idea of a special treatment for CPP blocks and developed a maintainable tool incorporating in functions only. The revised version[31] is based on COCCINELLE — a tool which is predestined in modifying Linux source files.

[31] The complete source of this implementation is annexed in Listing A.7.

Chapter 4

Evaluation on ARM Platforms

To show the broad applicability of the new approach in various real-world use cases, I evaluate FLIPPER on two distinguishing devices based on the ARM architecture in four different scenarios and compare the results with the baseline kernel.

The Raspberry Pi is my first platform for evaluation: With over 3 million delivered units[32] it is probably the most popular low cost mini computer on the market. While it is used in very different purposes this evaluation tries to cover a part of the variety by selecting three distinct situations:

- Using the Raspberry Pi as media center running "raspBMC"
- Learning to write web browser applications on "Coder"
- Setting up a wireless access point which makes the user's web traffic anonymous by routing it through the TOR [61] network (called "OnionPi")

In contrast to the resource-constrained mini computer, the second part is focussing on a high-end smartphone: The LG E960, also known as Google Nexus 4. This choice allows the demonstration of the approach on high-performance devices with more specific hardware and higher throughput due to its multicore processor. The smartphone with the development version of Ubuntu Touch as the operating system is used in a typical manner including making calls, taking pictures and data exchange with external devices.

The overall structure of the trace test runs remains the same, which I denote as *twenty minute approach*: After booting the device with a prepared kernel, it is allowed to settle for ten minutes to avoid potential inferences of any initialization code run after startup. During the next ten minutes the use-case-specific actions are performed manually in a pre-defined schedule. Afterwards, the backup of the trace file and the shutdown is initiated automatically.

[32] Source: `http://www.raspberrypi.org/raspberry-pi-at-buckingham-palace-3-million-sold/`

Note: For a fair comparison, the term "features" denotes only binary and ternary KCONFIG features in this chapter since the rare value features are not handled by the VAMOS tools. A detailed documentation of the results for each test cases with respect to these items can be found in in the Appendix B.

4.1 Raspberry Pi

To evaluate the effectiveness of the proposed approach, I generate a configuration from the data collected by FLIPPER and measure the reduction achieved in terms of KCONFIG features, text segment size and the number of source code lines compiled compared to the baseline kernel.

I performed all trace (and verification) runs on identical Raspberry Pi in hardware revision 2 (2011.12) with 512 MByte memory[33], the operating system and userland was transferred on 16 GByte Class 10 SD cards. The following steps were performed on a 16 core[34] (Intel Xeon E5620) server machine with 24 GByte memory – but in principal any current desktop machine model with a similar amount of memory could do as well, of course. Mapping the bitmap to source code locations, correlating these to configuration items and generating the solution with the given setting takes around 10 minutes, with the latter part taking most of the time.

I was able to successfully boot the tailored kernels, after I put 14 test case-independent features onto a whitelist (listed in B.2), which I identified manually by comparison with the original configuration. This was less tedious than it sounds, as the items provided were mainly specific to the hardware (for instance, to bypass ARM errata) or other low-level features that I could identify by their name.

4.1.1 Coder

Google developer Jason Striegel published his open source project Coder [13] in September 2013[35], which turns the Raspberry Pi into an educational web developer platform assisting in learning HTML, CSS and JavaScript: The mini computer acts as a server providing an easy to use web-based application manager and editor with a few sample apps. For the evaluation, I used version 0.4, which comprises a Linux kernel 3.6.11. As the system is running as a server and only used via network, no keyboard or screen were connected; the only external cable besides the power supply was an Ethernet cable (RJ45).

[33] Full hardware specification can be found at B.1.

[34] In fact, multiple cores are not necessary at all for this step: The tool is only running single threaded!

[35] Official announcement in GoogleDev blog: `http://googledevelopers.blogspot.de/2013/09/coder-simple-way-to-make-web-stuff-on.html`.

The schedule (Figure 4.1) was quite simple: I connected to the service after ten minutes of idling, changed some of the code provided in the default installation package and executed a web application.

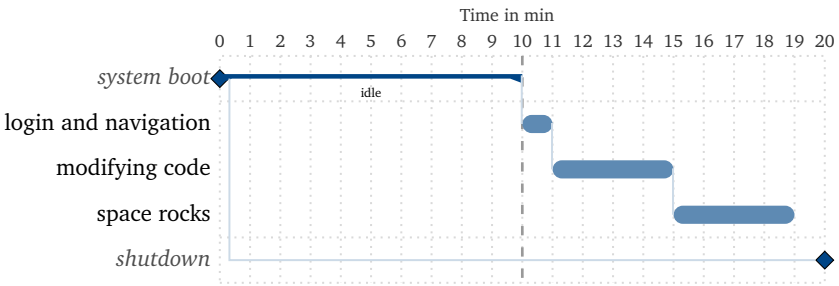


Figure 4.1 – Schedule for collecting addresses at the Coder scenario on Raspberry Pi

The results provided in Table 4.1 show that the number of enabled KCONFIG features is reduced by about 74 percent, leading to a text segment by almost a fifth of its original size. Using DWARF debug information, I also determined the number of source code lines actually compiled into the kernel. The reduction is similar to the other metrics, with savings reaching more than 70 percent.

Metric	Baseline	Tailored
KCONFIG features	1,678	429 (25.6%)
Text segment (byte)	22,621,072	4,835,648 (21.4%)
Source code lines	845,627	239,680 (28.3%)

Table 4.1 – Results for the Coder scenarios using three metrics. Percentages shown are quotients between the FLIPPER tailored version and the corresponding original configuration file

Using the tailored kernel, I was able to use all functionality provided by Coder: Modifying code on the web interface as well as running the sample applications worked perfectly. Connecting additional devices (not used in the tracing scenario) like monitor or keyboard has (as expected) no effect since these drivers are removed during the tailor process.

This lead me to the question: How do uninitialized hardware components influence the power consumption?

To answer this question, I employed a digital multimeter[36] which allows me to measure (and record) the electric current of the USB power supply by using a constant voltage of 5 V DC.

[36] HAMEG HM8012 with a DC current measurement resolution of 1 mA, connected to a PC using the RS-232 interface.

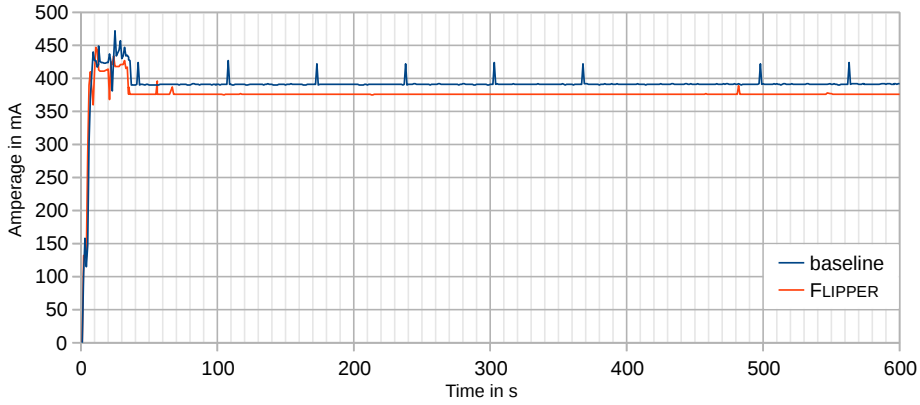


Figure 4.2 – Comparison of power consumption between original and tailored kernel in the Coder scenario

Comparing the power consumption (Figure 4.2), I was able to observe reductions of around 1–2 percent with my tailored kernel: While the baseline kernel has an average consumption of around 391 mA after finishing boot (with frequently occurring amplitudes of 426 mA), my tailored kernel needs about 376 mA in this stage (with less amplitudes). Although there is a very slight improvement, I do not think this is really a notable difference. Therefore I decided to refuse a detailed investigation in the later scenarios as long as a quick examination does not indicate a significant change.

4.1.2 OnionPi

The second scenario employs the Raspberry Pi as a proxy for the TOR anonymity network. This is done by installing the TOR client software on top of a standard Raspbian Linux distribution using the Linux kernel version 3.6.11. The OnionPi was set up according to instructions[37] provided by ADAFRUIT [36], a company operating an online platform (and selling enhancements) for educational electronics like the Raspberry Pi.

Connectivity to the internet is provided via the Ethernet port, while a miniature USB wireless adapter[38] is used to establish a WiFi network. Traffic sent through this network will subsequently be routed via TOR.

To reconstruct normal usage, a computer connected to the WiFi network after the settling phase, visited web sites using a browser and fetched emails from a server. After five more minutes, a smartphone logged into the network and was then used to visit web sites. A graphical representation of the schedule is provided in Figure 4.3.

[37] The version and available software packages of September 27th, 2013 were used.
[38] Model *EW-7811UN* by *EDIMAX*, supporting IEEE 802.11b/g/n.

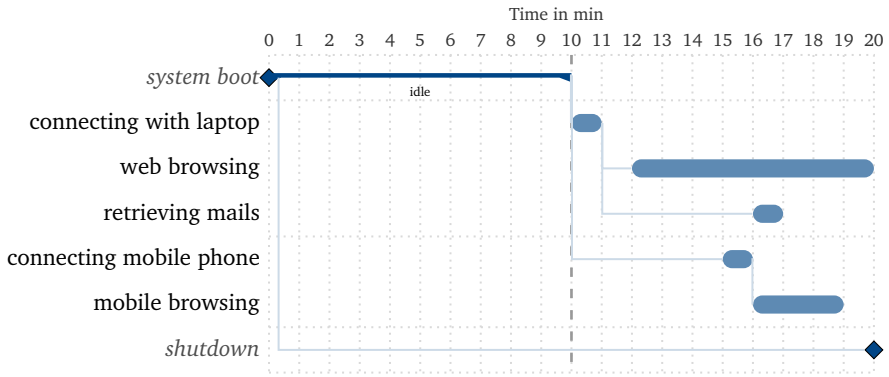


Figure 4.3 – Schedule for tracing OnionPi on Raspberry Pi

The results for the tailored Linux kernel are provided in Table 4.2. As with the previously presented test case, the number of features present in the tailored configuration file is reduced to a fourth, the text segment shrinks to 22 percent its original size and the number of source code lines mentioned in the DWARF debug information is decreased to less than a third.

Metric	Baseline	Tailored
KCONFIG features	1,678	426 (25.4%)
Text segment (byte)	22,688,201	5,041,604 (22.2%)
Source code lines	846,554	252,362 (29.8%)

Table 4.2 – Results for the OnionPi scenarios using three metrics. Percentages shown are quotients between the FLIPPER tailored version and the corresponding original configuration file

The tailored kernel was tested with the schedule again and provided the same functionality as before without any problems or noticeable performance degradation. Additionally, I let the Raspberry Pi provide a WiFi hotspot in the departments laboratories for a period of over two weeks. Daily use with various devices proved the tailored system to be stable and to perform without any problems in a realistic environment.

4.1.3 RaspBMC

In this scenario, which resembles the very common usage of the Raspberry Pi as a media center, the Raspberry Pi is connected to a screen via HDMI, speakers are plugged into the audio port, internet connectivity is provided using Ethernet and a USB keyboard[39] (with media key extension) is used to handle the machine. I used

[39] *CYA Model 210XX* by *Cherry*

the December version[40] raspBMC, running on a Linux kernel 3.10.25. The available extra video decoding hardware (both MPEG2 and VC-1) is enabled by adding the corresponding license keys to the Raspberry Pi boot configuration.

After the settling period mentioned earlier, I first started an integrated app to show the current weather. Subsequently, a video clip was streamed from a remote SFTP server, followed by multiple accesses to the web front end for remote controlability. Lastly, two more video clips were played. A detailed description can be obtained from Figure 4.4.

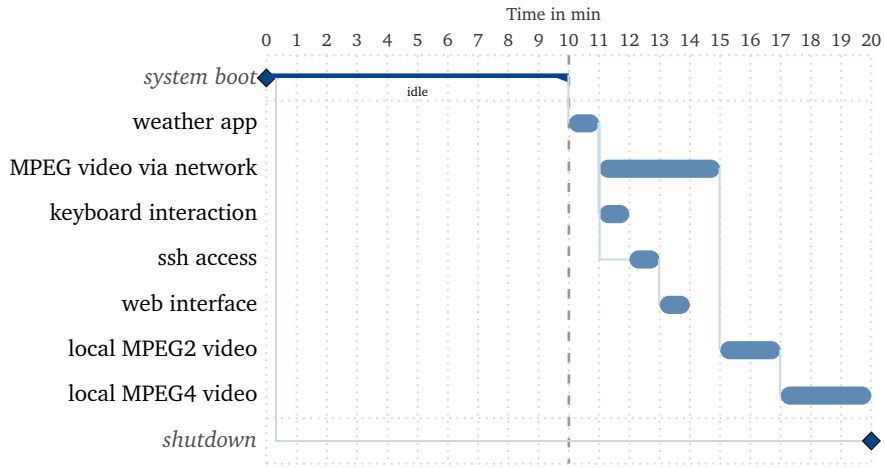


Figure 4.4 – Schedule for tracing raspBMC

In contrast to the two use cases presented before, the tailored kernel was not fully functional out of the box: Parts of the frame buffer device[41] are configured as modules — but for successful compilation these options must be statically included. Since this seems to be a bug in the Linux KCONFIG model, I manually corrected these options and successfully continued with the process.

Metric	Baseline	Tailored
KCONFIG features	1,819	452 (24.8%)
Text segment (byte)	22,960,278	5,656,040 (24.6%)
Source code lines	842,460	275,403 (32.7%)

Table 4.3 – Results for the raspBMC scenarios using three metrics. Percentages shown are quotients between the FLIPPER tailored version and the corresponding original configuration file

[40] Announcement and further information at `http://www.raspbmc.com/2013/12/raspbmc-december-update/`.

[41] Configuration options `CONFIG_FB` and `CONFIG_FB_BCM2708`.

The results provided in Table 4.3 show that the number of enabled KCONFIG features is reduced by over 75 percent and the number of lines compiled into the kernel is reduced by more than two thirds, leading to the total size of the text segment less than a quarter of its original size.

Using this generated kernel, I initially tested its functionality by running the tasks from the workload description again. I was not able to detect any degradation in performance or usability and could also use features provided by raspBMC I did not trigger during the observation phase. When I subsequently handed out one of the systems running on a tailored kernel to fellow researchers, they did not experience any problems during daily private use as a media center over the course of four months.

4.1.4 Comparison with FTRACE

When I tested the different approaches, I found FTRACE being capable of collecting enough addresses to compile a usable Linux kernel. Thus, I also generated configurations for all scenarios using the FTRACE collection method. While the kernels produced were able to boot into the scenarios[42], and the resulting configurations were even smaller (see Figure 4.5 for a quick comparison, details for each use case are attached in B.1), a manual comparison showed that especially during boot a lot of information was lost due to the high load induced by the FTRACE data collection mechanism. However, the kernel configuration system was luckily able to recover most[43] of the required configuration options.

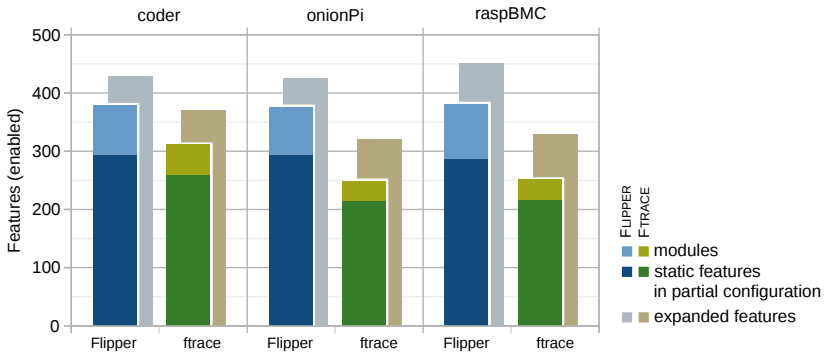


Figure 4.5 – KCONFIG feature selections for the Raspberry Pi test cases when using different data collection methods

[42] Only at the raspBMC use case I encountered a problem while tracing: Since the Linux kernel version 3.10 produces many identically function calls at boot time, the UNDERTAKER tools in default configuration are not able to process them as fast as necessary, the system will get stuck. Only setting very short flush cycles for the FTRACE ignore module (about 100 times more flushes than the default one) will allow the system to start up.

[43] One restriction I discovered so far is, a tailored raspBMC kernel is not able to shut down the system using the graphical user interface — in contrast to the new FLIPPER approach.

The problem is examplarily shown in Figure 4.6 for the raspBMC use case described above. During start up and for over five more minutes in the settling phase, the number of observed code points increases continuously. After this, execution of the scheduled actions clearly shows the detection of additional functions and distinctly visible increases in enabled KCONFIG features. Analysing the same scenario using the new FLIPPER approach, I find a very different situation: While the functionality triggered by the defined actions from the schedule can still be seen as a very slight increase in the number of code points recorded, the configuration generated is already almost completely stable from the beginning of my recordings (in both cases, snapshots of the current tracing progress were collected as early during the upstart phase as possible). The evolution of features is similar for all use cases I presented in this work; it is, however, not compulsive for every possible case. Nevertheless, while the measurement time frame is just long enough for the FTRACE approach to generate a working tailored kernel, FLIPPER delivers a more comprehensive solution much earlier during the observation phase.

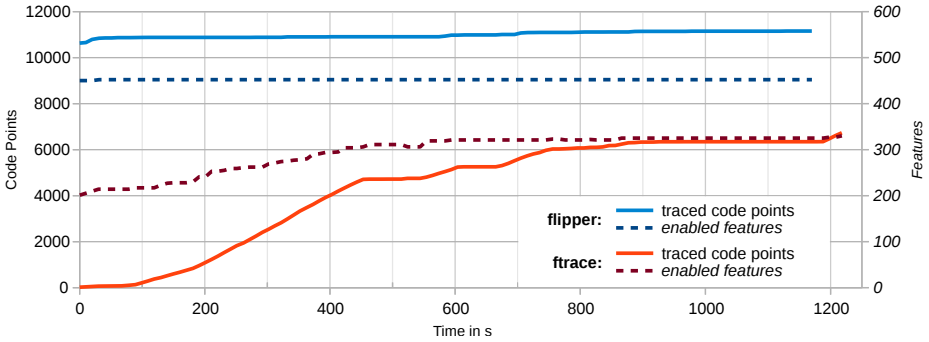


Figure 4.6 – Evolution of recorded points in the source code and KCONFIG features enabled in the resulting configuration for the raspBMC use case using both old and new approach

4.2 Google Nexus 4

When running on a smartphone, the need for configurability to support a lot of hardware vanishes: As almost no peripheral hardware can be connected, the kernel configuration will not need to provide drivers for them. On the other hand, a smartphone often uses very special hardware, making it hard for an engineer to derive a valid initial Linux kernel configuration. Additionally, some phones do not support SD cards to be inserted for more storage space, thus it would be good to have an operating system as small as possible.

4.2.1 Ubuntu Touch

Canonical, the company behind the distribution Ubuntu, announced a mobile operating system based on Linux in early 2013[44]: The Ubuntu Phone[45]. Started as a fork of the Android based CyanogenMod [14] it became a stand-alone mobile Linux distribution with distinctive features. Although currently no preinstalled Ubuntu Touch phones are delivered[46], the open access to the sources and the early stage of development supported my decision to choose it for further investigation with my approach. Due to the limited number of supported devices, I had to use a Google Nexus 4[47] for the evaluation.

The test load defined by the schedule (Figure 4.7) imitates everyday use of smartphones: After the initial waiting interval, the phone was first used to play some music stored on the device. Then the internal front and back camera were used to take pictures, WiFi was enabled and used by the web browser to load a web site containing a video. After that, one incoming and one outgoing phone call were initiated. Lastly, the phone was connected to a PC via USB and the images taken were transferred from the phone to the computer.

As the Google Nexus 4 was the main development platform for Ubuntu Touch, I presume the developers already have invested a lot of time to reduce the number of activated KCONFIG features. Consequently, the number of enabled features in the baseline configuration is already more than 35 percent lower than in the kernels provided for the Raspberry Pi. Therefore I assumed, my approach would not be able to achieve a similar level of reduction in terms of enabled KCONFIG features as in the Raspberry Pi case.

The results are shown in Table 4.4. As expected, the number of enabled KCONFIG features is reduced by only 33 percent, thus lessening the text segment size by 17 percent and the number of source code lines compiled by 12 percent.

Metric	Baseline	Tailored
KCONFIG features	1,119	752 (67.20%)
Text segment (byte)	14,464,220	12,037,224 (83.22%)
Source code lines	564,324	494,082 (87.55%)

Table 4.4 – Results for the automated tailoring of Ubuntu Touch on a Google Nexus 4 smartphone.

[44] See `http://blog.canonical.com/2013/01/02/its-official-ubuntu-now-fits-phones/`.

[45] During the development the project was promoted with different names like “Ubuntu Phone” and “Ubuntu Touch” (which both had multiple meanings), while the — currently discontinued — project “Ubuntu for Android” was promoted as “Ubuntu for Phones” in the beginning. For clearance I denote the mobile operating system simply as “Ubuntu Touch”.

[46] According to Canonical Ubuntu Touch based phones are shipped in 2014: `http://insights.ubuntu.com/2014/02/01/mwc-2014-online-press-pack/`

[47] The full hardware specification can be found at B.2.

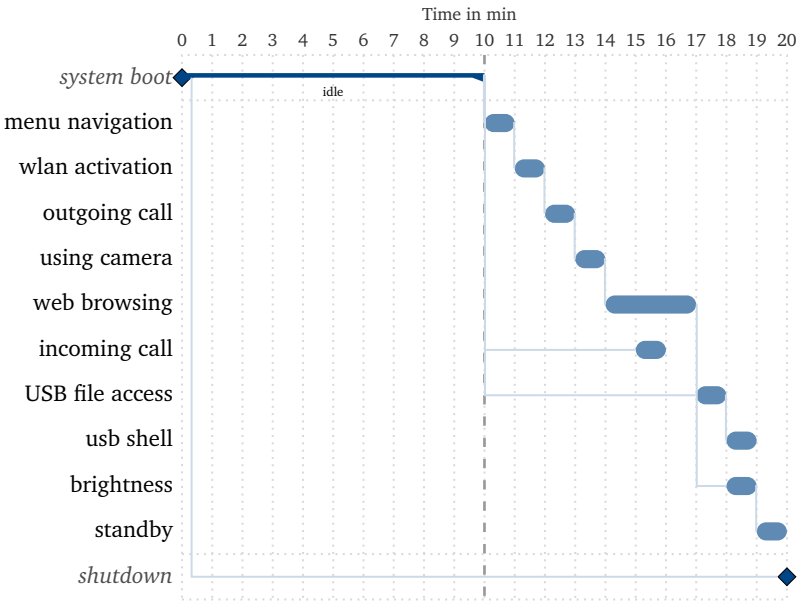


Figure 4.7 – Schedule for tracing Ubuntu Touch on Google Nexus 4

The tailored kernel was then used for the same purposes as described in the schedule, and performed flawlessly. Furthermore, it was possible to use previously untouched functionality: I was able to send and receive text messages, which was explicitly not part of the test load — this might be an evidence for the coarse-grained configurability of drivers[48].

While the reduction is not as high as for the Raspberry Pi use case, my approach is able to slice a third off the number of enabled configuration items. This result could provide valuable hints to the developers what additional features could be removed.

4.2.2 Comparison with FTRACE

As for the Raspberry Pi, I tried to generate a tailored kernel using FTRACE. On the Google Nexus 4, however, FTRACE simply produced way too much output: The heavy load generated by the continuous evaluation of the FTRACE output pipe most of the times lead to a watchdog being triggered, effectively breaking boot and my measurements.

In the rare cases the system was able to boot, the collected data was insufficient, as too much information was lost due to the limited buffer size of FTRACE: To make a generated partial solution bootable, over 180 KCONFIG features – more than a fifth of the total number of activated features — had to be added through the whitelist

[48] This topic is discussed in 6.4.1.

mechanism, rendering FTRACE practically unusable for data collection for even only an approximation of an automated solution.

4.3 Summary

Both on the Raspberry Pi and on the Google Nexus 4 smartphone the new FLIPPER approach is able to clearly reduce the kernel size — without any limitations on the usability of the particular scenario. About three-quarters of the baseline kernel size can be cut off in the Raspberry Pi use cases. Although the former FTRACE-based approach can even enhance this reduction with almost all required functionality, this is only possible due to the favorable circumstances like the extensive idle time. In the case of the Ubuntu Touch on a Google Nexus 4 smartphone only the new FLIPPER approach is applicable; I was able to reduce the number of features to about two-thirds compared to the baseline configuration without influencing the necessary functionality of the workload.

Chapter 5

Emulation Framework for Approaches

I demonstrated the applicability of the new approach in different, manually operated real-world scenarios. However, the VAMOS/CADOS UNDERTAKER development toolchain lacks an automatically evaluation environment for the tailoring tools — therefore I decided to address this subject. For regression testing and a better comparability between the different approaches the initial state as well as interactions with the target system have to be as similar as possible. This can be achieved with a customizable virtual machine with pre-defined automatic simulation of input. In addition, such a standardized test suite will offer the ability to verify future development in the UNDERTAKER tailor project. In this chapter I first draw the development of the framework, afterwards I present an evaluation concerning all approaches.

The workflow of the emulation framework shown in Figure 5.1 is completely automated: At ❶, required kernel modifications are performed and the baseline configuration is enhanced (if necessary for the specific approach). The compilation step ❷ ships the kernel as a compressed package, which is then installed in the ❸ virtual machine. After an idle phase, the scheduler executes predefined interactions on the server. According to the particular approach, the code points are resolved from the trace file and solved by UNDERTAKER in step ❹. Then the building steps are repeated again with the tailored configuration — ❺ building the kernel and ❻ running it in a virtual machine — but without tracing this time. If the interaction with the tailored kernel returns the same output as the original one, the tailored configuration is considered fully functional (step ❼).

Although this framework should be able to cover different use case scenarios, I revived the web server set up described in previous work for this thesis: A client performs multiple page requests to the web server and verifies the response. To simulate administrator access, the client additionally executes a few commands via a remote shell. A detailed description of the complete schedule is attached in B.10.

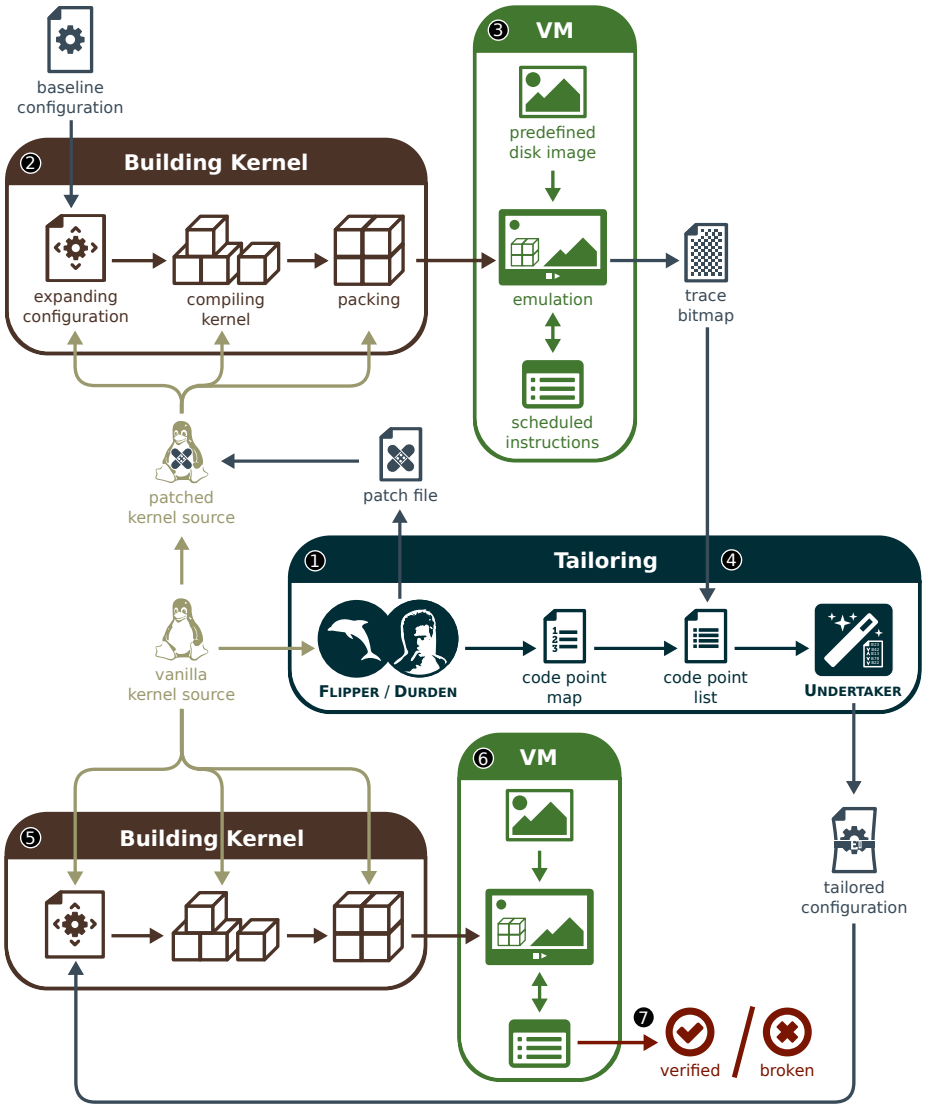


Figure 5.1 – Schematic representation of the new approach' emulation workflow

As an additional comparison, the framework performs an extra run without any UNDERTAKER tailor modifications in the virtual machine while the emulator records the code points in order to provide the best possible trace results which can be achieved: Every address is recorded while this approach allows side effect free analysis[49] at the same time. This — labelled EMUTRACE — allows me to draw further conclusions concerning untraceable features and their whitelist items.

[49] Unlike the UNDERTAKER tool, which is employed as a userspace application during the FTRACE approach and therefore produces side effects.

5.1 Environment for Virtual Machine

Several systems provide extensive emulation support. Besides QEMU system-mode emulation with its support for the kernel virtual machine (KVM) the tools BOCHS, VMWARE and VIRTUALBOX are perhaps the most popular ones. For the implementation of the program counter logger, the source code must be accessible. Except for VMWARE, all tools are available in an open source version. With respect to performance, the support of hardware virtualization should be favoured. Aside from BOCHS, this is supported by most implementations. The final decisive feature which convinced me of using QEMU/KVM was its extensive integration in Linux, enabling an easy utilisation.

Although my framework should be able to be extended to any Linux kernel and userland, I chose the Debian distribution as the primary target because of its popularity[50]. To be future-proof, I decided to deploy the testing release "Jessie". As the UNDERTAKER tailor tools were just supporting Ubuntu, I had to expand them on the functionality of preparing Debian systems[51].

Copy-on-write images allow an identical starting point. I have manually installed the initial system, containing — besides the minimal Debian Jessie base system — the Apache 2 web server [58], which operates with the PHP5 [40] script language. A PHP implementation of the prime number algorithm "Sieve of Eratosthenes" is responsible for producing workload. No graphic is configured, remote access to the server is granted by HTTP, HTTPS and SSH (with password authentication only).

The TOOL COMMAND LANGUAGE [57] (better known as TCL) extension EXPECT [18] allows quick scripting of automated interactions using common tools[52], which are the key requirements for a comparable simulation.

Since the other VAMOS tools' daily verification is managed using the continuous integration (CI) tool JENKINS [25], an integration of this framework into the research groups JENKINS instance suggests itself. Unlike the other projects strict hardware requirements need to be taken into account: Due to limitations of the SAT solver the target machine needs to have enough free memory (more than 20 GByte) for solving extensive trace files.

[50] Since it is hard to measure distributions popularity, there are no reliable sources. However, according to websites like `http://distrowatch.com/dwres.php?resource=popularity` a very high popularity of Debian (or Debian-based) distribution can safely be assumed.

[51] Consequently, the tools are now finally able to cover all three main initialisation systems: Beside UPSTART [62] now SYSVINIT [53] (for the Raspbian mentioned above in 4.1) and SYSTEMD [52] (new in the upcoming Debian Jessie release like announced (as voting result) on the official mailing list, see `https://lists.debian.org/debian-ctte/2014/02/msg00405.html`).

[52] In detail: WGET, CURL, SSH and SCP are utilized.

5.2 Emulator-based Code-Point Recording

Although the QEMU project is becoming quite complex (with over 1 million physical lines of code[53]), the very basic procedure of the dynamic binary code translation remains clear: For each piece of guest code to be executed, the cache is queried for a decoded host code representation. Without a representation, it is read and analysed block wise (that means until the next instruction performing a jump or modifying the host CPU state) [3]. The following steps have changed in detail by switching from DYNGEN to TCG (tiny code generation) due to performance issues[54], but both implement a way of translating the block to host compatible code and executing it. A good working point seems to be the first part of this procedure: Each time a new translated block is generated, the program counter (PC) has encountered an untranslated block and can be taken into the trace file. Certainly this will include the PC from userland applications (outside of the kernel address space), but this is not a problem at all: I simply configured the UNDERTAKER tailor tools to ignore invalid addresses.

Because of the dynamic memory allocation, I cannot support loadable kernel modules in this approach. Though it is conceptually possible to figure out the memory position and recompute the code positions, I decided to not carry out this extension due to its limited application: Just disabling module support appears to me as a practicable solution for this scenario.

Instead of enhanced modifications using an internal set which manages all logged addresses, I prefer a simple solution for the new EMUTRACE to ensure backward and (hopefully) future compatibility[55]: My QEMU semantic patch (shown in B.1) enhances the corresponding function to print relevant data to the standard output stream. Due to the fact that QEMU entirely flushes the 16 MByte or 32 MByte[56] cache (depending on the version), multiple occurrences of the same address will certainly happen. However, this is neither a problem for the UNDERTAKER tool nor a remarkable performance impact: Tests in the simulation process demonstrated that the output including duplicates is just twice the size of a distinct address list. Finally, I simply recycle the address lookup tools created for the FTRACE approach to retrieve the position in the code: I obtain the required information employing ADDR2LINE in association with a debug kernel, the latter tools in the chain are used in their usual manner.

[53] Code evolution visualized at `http://www.ohloh.net/p/qemu/analyses/latest/languages_summary`.

[54] According to the latest QEMU INTERNALS MANUAL [42] '2.2 Portable dynamic translation'.

[55] In fact, my patch is applicable for all versions in the last 10 years since Bellard added the 'precise self modifying code support' (Git commit: `http://git.qemu.org/?p=qemu.git;a=commitdiff;h=d720b93d0bcfe1beb729245b9ed1e5f071a24bd5`) — even though there was a code refactoring in 2012: `https://lists.gnu.org/archive/html/qemu-devel/2012-12/msg00407.html`!

[56] According to the QEMU INTERNALS MANUAL [42]: '2.5 Translation cache'.

First tests pointed out that this approach is not as applicable for appropriate comparisons as expected: The other approaches (the code injection based as well as the FTRACE based ones) ignore triggering functions involved in module initialisation, but the EMUTRACE logs — without any exception — every block. To cure this problem, I have extended EMUTRACE by a new routine to detect and remove code points which are involved in the module initialisation process[57] from the trace file afterwards[58]. For a better understanding the extended version is called EMUTRACE (NO INIT).

5.3 Scope of Evaluation

Although flexibility was one of the design concepts of this framework and many architectures can be used[59], this evaluation intentionally focuses on the x86-64 platform: Since the FTRACE approach was originally designed for this architecture, I achieve the best preconditions for a comparison.

This work focuses on the Linux kernel versions released within the last year (from version 3.10[60] to 3.15[61]) and employs the latest third party tools involved in the workflow available: UNDERTAKER in version 1.5, QEMU in version 2.1.0 (both updated daily from their development repositories) and the release candidate 21 of COCCINELLE 1.0. The virtual machine was configured with 4 GByte RAM and a 4 core symmetric multiprocessor system without graphic or sound support.

Although the schedule is very similar, some specific settings and interactions depending on the approach must be performed:

DURDEN is the prototype version of the new concept. For the evaluation, the block injection described in 3.2.4 is disabled due its drawbacks. At the end of the simulation, the map is gathered using remote file transfer (SCP).

FLIPPER denotes the revised (final) version of the new approach. Its results differ slightly from the prototype because of the limitations of COCCINELLE (described in 3.3.3). Besides this code patching, the framework handles tailoring with the final version in the same way like the prototype.

[57] Functions referred by the `module_init()` macro.

[58] However, since the detection only covers the static initialisation functions itself — but neither helper functions nor CPP macro generated initialisation routines —, this is not a complete removal and therefore it may lead to over-approximate results.

[59] Depending on QEMU — which currently supports over two dozens hardware targets according to its documentation [41] (1.1 Features).

[60] Published June 30th, 2013; release message `http://article.gmane.org/gmane.linux.kernel/1518301`.

[61] Published June 8th, 2014; release message: `http://article.gmane.org/gmane.linux.kernel/1720707`.

FTRACE based approach (described in previous work). The framework installs the UNDERTAKER tracing tools remotely according to the TailorHowTo [54].

FTRACE (EARLY BOOT) is the enhanced version utilizing special kernel boot parameters to start tracing in an early stage of system start up. Although the amount of additionally logged addresses is strictly limited[62], it provides better results compared to the FTRACE approach. Additional to the FTRACE set up, the kernel parameters are modified in the boot loader.

EMUTRACE, like described above, requires no interaction within the virtual machine, but for the emulation a modified version of QEMU (without KVM support) is engaged. Due to the slow software emulation a significantly higher idle time after boot is granted.

EMUTRACE (NO INIT) is identical to EMUTRACE concerning the trace file creation — only an additional step removing `module_init()` entries is appended afterwards.

5.4 Automatic Generation of Whitelists

Instead of generating whitelists with domain knowledge, I am now able to automate its generation employing this environment (according to the trail and error principle): At first, a complete run is performed without any whitelist involved. If the resulting kernel runs well, no whitelist is necessary. Otherwise, every feature item only present in the original configuration[63] (but not in the resulting) requires

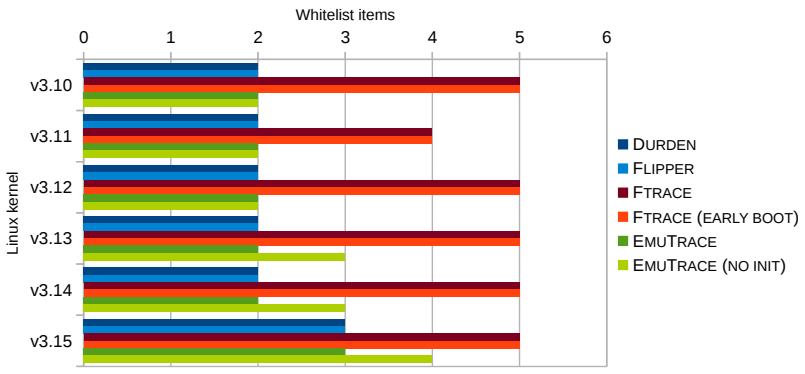


Figure 5.2 – Necessary items in whitelist depending on method and kernel version

[62] Due to its design as a ring buffer and memory constraints, its usability depends on the scenario.

[63] In most cases this will result to several hundreds of items. However, it seems to be sufficient to use a previous successfully tailored configuration for comparison. This trick will lead — depending on the quality of the previous tailoring — to only a dozen of items left and can be processed in a few hours (instead of weeks)!

a detailed evaluation: For each case a temporary whitelist is constructed, containing all items missing in the new configuration but without the selected one. If the later steps (tailoring, compilation, simulation) are successfully performed, the selected item is not necessary for the whitelist and is dropped.

Of course minimizing of an existing, perhaps over-approximated whitelist to only necessary items can be done in a very similar way, just by testing each item on the whitelist in the way described before.

configuration feature	count
CONFIG_UNIX	36
CONFIG_BINFMT_SCRIPT	36
CONFIG_RD_GZIP	16
CONFIG_DEVTMPFS_MOUNT	12
CONFIG_INOTIFY_USER	10
CONFIG_OPTPROBES	3

Table 5.1 – Accumulated occurrence of whitelist items after minimization in the emulation framework for every version and approach (in total 36 whitelists)

Applied to the relevant Linux kernel versions above, I am able to minimize the whitelist — depending on the case — down to only one or two entries. The fairly evenly sizes in Figure 5.2 (listed in detail at B.3) seem to be no coincidence at all: Only 6 distinct whitelist items are present in all cases (listed in Table 5.1). While CONFIG_UNIX and CONFIG_BINFMT_SCRIPT are needed independently of approach or version, items like CONFIG_DEVTMPFS_MOUNT and CONFIG_INOTIFY_USER are only (or, in case of CONFIG_RD_GZIP at least mainly) used in the FTRACE variants.

5.5 Evaluation of Test Series

I performed every approach on each kernel version with the corresponding whitelist. Every run was able to generate a fully functional tailored kernel on the first try, however the amount of reduction varied markedly between the different collection approaches.

Method	collected addresses	code points	static features	modules	summarized features
Baseline			1,402	2,741	4,143
DURDEN	13,860	13,860	674	97	771 (18.61%)
FLIPPER	13,064	13,064	673	96	769 (18.56%)
FTRACE	7,731	7,009	421	17	438 (10.57%)
FTRACE (EARLY BOOT)	7,388	6,767	435	17	452 (10.91%)
EMUTRACE	14,128,732	84,577	988	2,502	3,490 (84.24%)
EMUTRACE (NO INIT)	20,284,134	76,962	931	1,706	2,637 (63.65%)

Table 5.2 – Collected data and the resulting features by automated tailoring with different approaches (Linux kernel v3.15), compared to the Baseline.

EMUTRACE logs more than 14 million memory addresses[64] in the latest Linux kernel v3.15, which can be resolved using debug information to almost 85,000 unique code points (Table 5.2). Although the number of collected addresses is a thousand times higher than the other approach, these values should not be overrated: Only the code points are really important for the further processing, which is only about six times higher than FLIPPER. The feature reduction is only about 16 percent, while the removal of `module_init()` in EMUTRACE (NO INIT) clearly improves the result to about 36 percent. However, the modules still clearly remain the main course for this insufficient tailor result[65]. The new approaches achieve a decrease of about 81 percent compared to the baseline configuration, while the FTRACE based approaches both produce a set containing only a ninth of its original items — a reduction of 89 percent.

Method	code size (in bytes)	compiled C lines
Baseline	73,499,239	2,494,433
DURDEN	9,865,887 (13.42%)	394,160 (15.80%)
FLIPPER	9,866,461 (13.42%)	394,333 (15.81%)
FTRACE	5,574,146 (7.58%)	233,056 (9.34%)
FTRACE (EARLY BOOT)	5,678,067 (7.73%)	237,098 (9.51%)
EMUTRACE	53,297,546 (72.51%)	2,226,537 (89.26%)
EMUTRACE (NO INIT)	41,454,897 (56.40%)	1,735,278 (69.57%)

Table 5.3 – Comparison of tailored kernel binaries (Linux kernel v3.15)

[64] In fact, this number can vary considerably in the emulation based approaches (the v3.15 EMUTRACE (NO INIT) lists 20 million memory addresses) — but in this case these are not only unique addresses: Due to the flush mechanism in QEMU (described in 5.2) the quantity differs much without having an effect on the resulting code points.

[65] Obviously code outside `module_init()` is executed during module initialisation — but since the EMUTRACE approach was created to identify the limitations of the general approach (and successfully does), I rejected to carry out a further investigation to solve this problem.

Comparing the binaries of the resulting kernel, I can observe a reduction to only 8 percent of machine code instructions at the FTRACE based approaches, while the new FLIPPER approach reaches about 13 percent of the original code size (cf. Table 5.3). The number of (unique) compiled C lines are with about 9 percent and 16 percent respectively in an equal proportion. However, the kernel created using the emulation approach is about 56 percent up to 73 percent the size of the baseline kernel (depending on the handling of initialisation functions), with 70 percent to almost 90 percent compiled C lines.

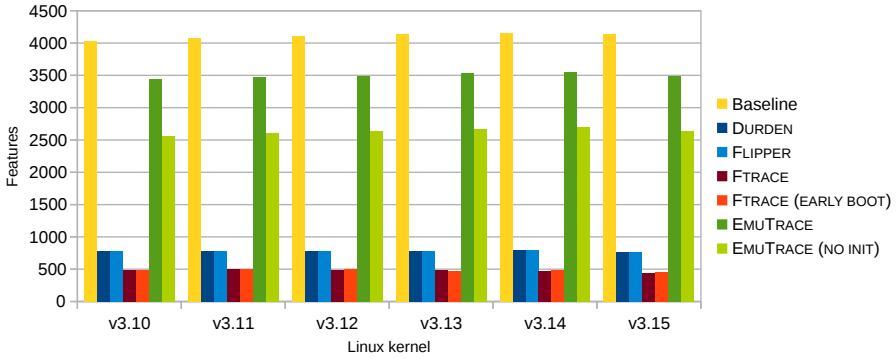


Figure 5.3 – Number of enabled features depending on approach and kernel version

Figure 5.3 depicts that the results for each approach are pretty static and fairly independent from the underlying Linux kernel version. Detailed information on this evaluation of additional versions is provided in B.3. In fact, I was not able to track significant changes in over 350 runs, neither did I find a constellation where even a single approach failed with a valid whitelist[66].

5.6 Summary

With employing QEMU and standard Linux tools, I established a framework which supports regression testing for further development and provides an easy to use interface for detailed comparison of the existing approaches. I successfully verified the test against all six Linux kernel versions published within the last year, effectively without manual guidance[67] since the framework has the ability to automatic generate (or minimize) whitelists, too. Due to the weak limitation of resources[68] in this

[66] The only problems I encountered were defects in the Linux variability model, which are not fixed in previous versions (like the missing dependency of `CONFIG_IRQ_DOMAIN` in `CONFIG_GENERIC_IRQ_CHIP` in v3.12 and earlier) — I solved them by backporting the bugfix patch.

[67] In two cases a manual intervention in the Linux kernel source was necessary since the older kernel versions had unresolved bugs.

[68] The resource settings of the emulation framework used for the evaluation of this chapter can be compared with a current desktop workstation: 4 GByte RAM and 4 core symmetric multiprocessing.

evaluation, the findings are comparable to the Raspberry Pi (described in Section 4.1): Kernels tailored with the FLIPPER approach are more extensive than those generated using FTRACE, but require less guidance (by whitelists) and computing time.

Since the software emulator interprets every instruction[69], I can achieve the probably highest accuracy possible in the general approach with the new emulator-based code-point recorder EMUTRACE. Together with an automatic whitelist generation/reduction script it discovered untraceable features — the only additional features necessary for tailoring a fully functional kernel with my new FLIPPER approach, whereas the former FTRACE approach requires additional guidance in the whitelist. However, due to the lack of a comprehensive removal routine for initialisation functions (discussed in 6.1), the high accuracy lead EMUTRACE to generate large kernel configurations — far too much for a fair comparison with the remaining approaches.

[69] Having the hardware acceleration KVM disabled, of course.

Chapter 6

Discussion

Although I successfully developed a code recording tool which deals with limited resources on embedded systems, I did not implement all of my initial objectives — like the manipulation of conditional blocks — into the final version. In the following section I will discuss the limitations and challenges of my approach in respect to existing techniques.

6.1 Accuracy

The completeness of data collected by FTRACE becomes significantly worse when aiming for smaller systems: Even if you are lucky and a tracing infrastructure is available on your target system, the low computing performance is a big issue: The slower the FTRACE output can be produced and parsed, the higher is the probability to lose potentially important functions which were executed. Setting a bit, on the contrary, will not affect performance as badly. In my test cases, the overhead induced was less than five percent. Nevertheless, I would like to point out that (as opposed to the old FTRACE version) the FLIPPER approach cannot be disabled at run-time: The small overhead will always be present during the observation phase.

The most obvious difference between the old and new approach is the tracing speed: FLIPPER almost instantly collects the code points, while the old approach needs several minutes (more than seven minutes in the raspBMC scenario for instance) until a saturation is reached and no new code points are triggered.

Another important difference is the point of time at which the collected data starts: Using FTRACE, you can in principal only collect data as soon as the file systems have been mounted by the kernel and an initialization script can be executed[70].

[70] An enhancement using the kernels ability of early boot FTRACE logging into memory slightly improves the number of traced code points like presented in the emulation evaluation (Table B.3). Because of the restricted amount of reserved memory, this finally causes just an increase of up to a fifth more data — with a very limited impact on the resulting configuration (only up to 3 percent new features).

This inevitably leads to missing data from the very beginning of the boot process, which provides important information about features corresponding to the hardware Linux is running on. This turned out to be the case: Using the FLIPPER method, which can effectively begin to collect data in the very first function the (unpacked) kernel executes, I was able to identify more relevant configuration options.

In comparison with the FTRACE approach, there is still another improvement in accuracy: Inlined functions (no matter if instructed by the `inline` keyword or not) are usually not accounted by FTRACE since the compiler performs the responsible profiling enhancements only after performing all code optimizing steps[71]. This is — besides the code points triggered at boot time — a reason for the increase of traced code points: For example, in the raspBMC use case the FTRACE approach identified about 6,700 called functions, while the FLIPPER method found more than 11,000 relevant places. Moreover, a relation between the whitelist and the number of trace points can be clearly observed in the emulation approach: While the new approach logs up to twice the number of real code points (and in the worst case at least 150 percent), the number of necessary whitelist items has clearly decreased.

This higher accuracy on the other hand had an unforeseen impact: When using a kernel without loadable kernel module support, Linux probes the devices. Thus, it will invoke the initialization routines for every driver very early during boot — even if the device itself is not present. For this case, an execution of the `module_init()` function[72] is not sufficient to determine if a driver is needed. But if more functions in the driver are called, the device is most likely present in the system.

While FTRACE is not able to detect these initialisation calls, I currently handle this situation in FLIPPER by excluding functions linked in the `module_init()` macro from being patched. For the EMUTRACE approach, the triggered functions are removed from the trace file before further processing continues, which results in the same outcome as the FLIPPER procedure.

If the initialization function calls other functions itself, they will still be registered in the FLIPPER bitmap and their configuration requirements will be accounted for in the generated configuration. However, I found the over-approximation in terms of enabled KCONFIG features to be reasonably small — the functions called by the initialization functions are mostly related to memory and data structure allocation — and, moreover, helpful to accurately detect more functionality being exerted during the test run.

But in the case of EMUTRACE, the tailored kernel size remains disproportionately high. Closer examination reveals that initialisation routines are still the reason: For

[71] Because of the GCC option `-fearly-inlining` (enabled per default), it is not possible to indicate such inlined functions in the optimized intermediate code — they appear to be a regular part of the code block.

[72] In fact, several initialisation hooks exist, but `module_init()` is the only one available for modules. For a detailed list of these macros take a look at */include/linux/init.h* of the Linux source.

example, the CPP macro `module_hid_driver`[73], a helper macro for registering human interface device (HID), is used in every HID driver. During compilation, this macro expands to a new initialisation function, which is executed automatically at boot — with the result that every HID driver from the original configuration is triggered by EMUTRACE and hence in the tailored configuration. Since I developed the EMUTRACE for comparison only, I refrain from expanding the removal procedure.

It becomes clear that approaches with higher accuracy need smarter routines to handle automatically called initialisation functions to prevent an over-approximated configuration.

6.2 Selection of Features

As can be seen from Figure 6.1, the Linux kernel generated using FLIPPER in the Raspberry Pi scenario has about a third more KCONFIG features enabled in its configuration when compared to the FTRACE result. The features contained in this set are either a result of the higher accuracy by detecting the inline functions or mainly used for low-level purposes[74].

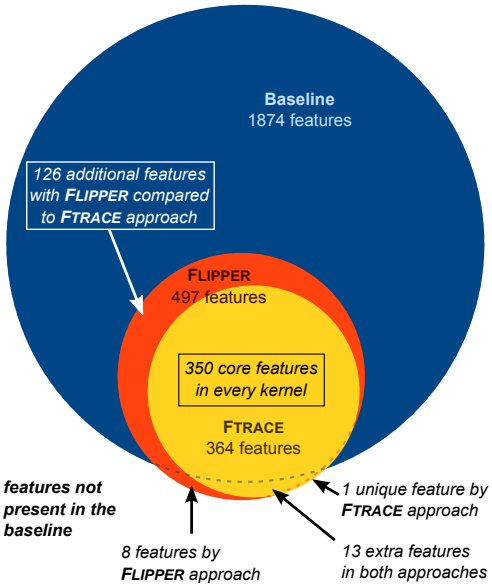


Figure 6.1 – Quantitative comparison of contained KCONFIG features (including value features) between the original kernel and tailored version in the raspBMC use case

[73] Defined in */include/linux/hid.h* of the Linux source.

[74] For example, they specify parts of the block device hardware support and other hardware probing routines.

A remarkable fact is that almost all features of the configuration created by the FTRACE approach are part of the FLIPPER solution — in contrast to the (expected) behaviour by comparison with the baseline: Since both approaches can only remove features, it is not surprising that their results are mostly a subset of the default configuration. The few features[75] enabled only in the generated configuration and not present in the original Linux kernel arise from the SAT solver approach: Some KCONFIG variables in the formula neither have been directly required during workload execution nor do appear in other features' dependencies. Thus, they will be seen as free variables; enabling or disabling them is at the SAT solver's discretion. One target for future work is to identify such free variables and provide guidance to the SAT solver; for example it could be instrumented to prefer the assignment present in the initial configuration file or to preferably consider options which optimize desired properties of the target system.

6.3 Granularity

One goal for the FLIPPER approach was to achieve a more accurate and fine-grained result for the feature-dependent blocks contained in the code, thus defining stronger dependency requirements and generating a configuration matching the use case more exactly.

Indeed, the majority of feature based code blocks is handled by conditional CPP directives, but since the possibilities of compiler optimization made their way into developers head, you can observe a slight opening for alternative feature handling. The usage of the `IS_ENABLED` macro[76] demonstrates this development: Introduced in Linux kernel version 3.1 it is used both in CPP and C code to evaluate feature dependent blocks[77]. It can be safely assumed that this development of using C conditions (instead of CPP ones) will keep increasing in the upcoming versions like shown in Figure 6.2. While this clearly enhances the sources readability, it does not only prevent an easy injection (without control flow analysis), but also requires a extensive enhancement of tools developed in the VAMOS project (especially UNDERTAKER).

Even if just `#ifdef` blocks are taken into account, it does not run like clockwork: As described in 3.2.4, I was confronted with uses of CPP in the Linux source code, where the insertion of additional instructions is a very hard task — these occurrences must be solved manually.

[75] A detailed list of the particular items and a short explanation is provided in B.1.

[76] Defined in `include/linux/kconfig.h`.

[77] It was figured out building a macro for both CPP and C code to check if an argument is defined or not is quite *tricky* — a solution posted to Linus Torvalds Google Plus code challenge `https://plus.google.com/+LinusTorvalds/posts/9gntjh57dXt` made its way directly into kernel since v3.4 (rc4).

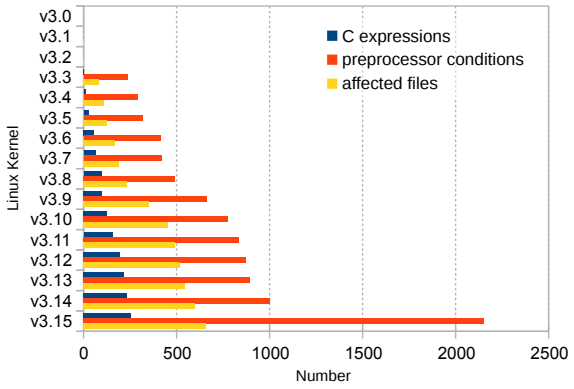


Figure 6.2 – Usage of the `IS_ENABLED` macro in the Linux kernel versions for the last three years

I filtered-out such problematic points in the code, ran the same test schedule and generated a configuration from this more exact approach. A first preliminary comparison revealed no notable difference to a configuration file obtained with only function entries patched.

From this, I conclude that conditional blocks inside a function's body do not contribute as much to the total variability as expected, therefore it is sufficient to collect data at a function level granularity; thus, my patching tool only inserts the bit-set operation into the beginning of every function definition encountered.

6.4 Completeness

During the observation phase, an application will most likely not trigger every single functionality it could. For example, certain errors and thus, execution of error handling code, might not occur during the test run, while they could arise during later, more extensive use of the tailored system.

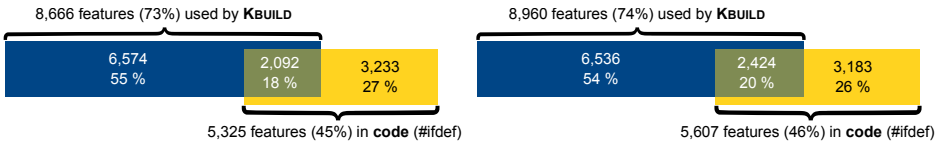
This is a principle problem of the approach: If you can only track events that are actually triggered, and no errors occur during observation, you can not prove that every functionality *possibly* required later will be included in the resulting configuration.

In practice, this problem seems to be less severe than it appears to be: In all of my test cases (including those from previous work of the VAMOS research group, where we tailored a server system and a workstation [55]), I did not encounter a single situation where any required functionality was missing — even though me and my colleagues have been using the tailored devices for a period of several months and exerted previously unused functionality such as sending text messages from the Google Nexus 4 phone.

Yet, there are also structural reasons that mitigate the potential risk of missing some important functionality during the observation phase, as shown below.

6.4.1 Use of Configurability in Linux

Linux mostly uses configurability in a way which leads to related, but possibly untraced functionality to be included during compilation: As mentioned in Chapter 2, more than 70 percent of the features in KCONFIG are used by KBUILD to determine whether a source file has to be compiled or not (see Figure 6.3); this particularily applies to drivers, where the corresponding configuration option will either include the whole driver for a device or leave it out entirely.



(a) v3.6.11 (used in OnionPi) **(b)** v3.15 (latest)

Figure 6.3 – Usage of KCONFIG features in different Linux kernel versions

This observation implies that in most cases triggering one single function inside a source file will be sufficient to have the whole compilation unit present in the resulting kernel, thus leading to the inclusion of additional unobserved functions, such as error handling code, from the same file.

In contrast, the 26 percent of KCONFIG features only present as CPP instructions implement fine-grained variability. As this technique is mostly used in the central parts of the kernel, missing functionality or inconsistency would already be detected as errors during link time or startup.

Note: The remaining features neither used by KBUILD nor in code (about 17 percent according to Dietrich et al. [17]) are KCONFIG internal meta features.

6.4.2 Test requirements

For special-purpose embedded systems, system developers typically have to provide test suites achieving very high or complete coverage of the system anyway (e.g., for certification purposes). Running these test suites as the workload during observation will greatly diminish the risk of missing, but possibly required code in the tailored kernel.

Finally, I would like to point out that the completeness concern would also arise if an expert manually tailors the system: How can the system developer be sure to have selected every configuration option required for his needs? Hence, I consider my automated approach as practically usable.

6.5 Untraceable and Alternative Features

I employ white-/blacklists to provide user guidance in situations my approach cannot cover. Since even EMUTRACE (which collects every code point) is not able to generate a working kernel without a whitelist, I can precisely name untraceable features. The detailed comparison of the runs performed in Emulation Framework for Approaches clearly shows that FLIPPER only needs a whitelist with these untraceable features — in contrast to the traditional approach, which requires additional items to succeed.

The usage of such lists, however, is not an issue: Selecting features necessary for a certain device can be done once (for example by the subsystem maintainer for this particular device or a distributor); like shown with a single whitelist for all Raspberry Pi scenarios it is not dependent on the use case.

It will also be much less work than manually getting a Linux vanilla kernel to work on a specific device. My tools can directly be used to simplify this process: When trying to determine features required for a new device, a developer could generate a configuration without using any lists and specifically search the difference between this preliminary configuration and the initial file for features relevant for the architecture or the specific use case. I used this approach to quickly determine the 14 architecture-dependent KCONFIG features provided in the Raspberry Pi use case (listed in B.2).

In theory, it is possible to automate the whitelist creation — although I present a proof-of-concept in Section 5.4, manual control might be necessary to achieve best results: Whitelists can be used for further guidance of the feature selection process, thus allowing domain experts to specify optional KCONFIG features they identified as being important for a certain system.

Particularly, for features presenting alternatives it might not be desired to simply use the (possibly randomly selected) option from the SAT solver, but rather to provide a choice known to be correct in advance — for example, for the memory allocator real free alternatives exist: Besides the default SLAB there are SLUB and SLOB. While every one has its own gain, the SLOB allocator might be predestinated for use in tailored kernel since it is designed to be memory efficient while having the smallest code base[78]. Once better alternatives are detected, experts can use whitelists to handle such preferences.

Features of string or numeric type (for instance, the kernel command line) are automatically taken from the original configuration and used after the SAT solver has generated an assignment for the binary features: Hence, the corresponding values in the tailored configuration are simply the same as in the distribution kernel.

[78] As announced at `http://thread.gmane.org/gmane.linux.kernel/344062`.

6.6 Impact on Non-Functional Properties

When optimizing an operating system for use on a deeply-embedded system, binary size is only one factor to consider. For example, reducing the power consumption of a long-running embedded device can be seen as highly important to lower not only the production cost but also the operating cost of a system.

I therefore also conducted preliminary measurements of the power consumption of the Raspberry Pi in the Coder scenario, but without significant changes. On the contrary, choosing from observation alone and employing a SAT solver to cover dependencies might lead to kernels with energy-saving features disabled.

One possible solution for a combined approach to optimize nonfunctional properties (i.e., power consumption) of the system as well as minimizing binary size could be the integration of heuristics as proposed by Siegmund et al. [47] for the impacts of KCONFIG features on desired properties into the selection process, thus guiding the approach to be more aware of these properties of the target system.

Again, this expert knowledge can currently be brought into the tailoring process by putting KCONFIG features previously identified onto the whitelist.

6.7 Dependency Modelling Defects

The fact that configurability is used for different purposes in the Linux kernel has lead to problems in the past [56]. This becomes an even bigger issue on the ARM architecture, with not only the architecture itself, but nearly every single device having different requirements. Additionally, in the ARM subtree many hardware peculiarities are modelled using KCONFIG. This has made ARM the by far biggest and fastest-growing subtree in terms of possible KCONFIG configuration options in the Linux kernel (Figure 6.4). Unfortunately, this also implies there is a higher probability certain things might be wrong or wrongly modelled.

Thus, it is extremely important for my new approach to gather as much information as possible: While some things (like the aforementioned `module_init()` functions) might lead to an over-approximation, I can overcome possible defects of the dependency model by supplying much more detailed data to the SAT solver, thus building stronger constraints and leading to a more solid solution.

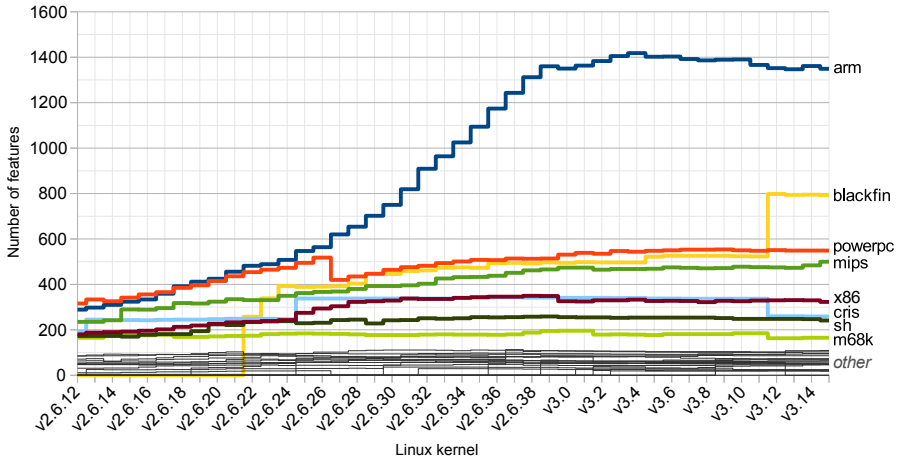


Figure 6.4 – Feature growth in Linux by architecture since 2006

6.8 Generalization beyond Linux

The approach presented in this thesis cannot only be applied to Linux, but can be transferred to other operating systems and software product lines.

The FLIPPER method to prepare the Linux kernel for data collection is directly applicable to any software project which uses the CPP to implement fine-grained variability, as it is only necessary to parse the source code and insert an instruction whenever a conditional block is found.

The harder part is the accurate extraction of models describing the features and their dependencies, which are required to find the correct mapping from the observations to their corresponding configuration items. Previous work [16], however, has shown the portability of the extractors I used for Linux to other software product lines such as the BusyBox UNIX utility suite [7] and the FIASCO microkernel [19], requiring only little effort.

Thus, the proposed method makes it feasible to generate small configurations matching an observed scenario for any configurable software product.

Chapter 7

Conclusion and Perspectives

Configuring system software for a given use case is a very challenging task. With hundreds of optional features to choose from, finding a small set of configuration options which includes just the *right* features is hard, even for a domain expert. This particularly applies to the Linux operating-system family, which offers nearly 14,000 configurable features.

For use on general-purpose computers, the solution provided by Linux distributors is to include as many features as possible into their kernel configurations, thus also increasing the size of the kernel. For the use of Linux in deeply-embedded systems, however, this is not an option: To keep costs at a minimum, as little memory as possible is to be occupied by the operating system.

While there are developers manually building small kernel configurations, these configurations often make assumptions of the usage of the embedded system which may not be valid for a specific use case.

Tackling these challenges, this thesis presents an automated kernel tailoring approach which can be used to generate a use-case–specific Linux kernel configuration, which is also suitable for use in resource-constrained embedded systems. As the resulting configuration might not take domain-specific knowledge into account, additional information can be brought into the generation process with minimal effort.

To enable regression testing and support future development, I stated an emulation framework with the ability to compare the new approach with existing ones by having an identical initial position and a well-defined workflow. Furthermore a modified emulator discloses the limitations in the basic concept of recording and evaluating code positions (like untraceable features).

My results show that for Linux, the kernel size can be reduced by up to 70 percent in real world scenarios. In contrast to the existing approach, the new process is clearly more resource-efficient and therefore applicable for resource constrained

devices. Results gained with FLIPPER can be used by system developers as a basis to easily create small, fitted software configurations for their systems, thus opening up a whole new field of use for Linux inside deeply-embedded systems such as control units in the automotive industry. Since the new concept basically has no additional requirements besides configurability, it can be transferred to any large-scale system software.

Future Work

Although I think the research in the CADOS/VAMOS group has explored a wide area of applications for the general approach to automatically derive a tailored kernel, there are starting points for future engagement left.

Using adjusted heuristics, it could be possible to identify more needless modules and to drive the reduction of code size even further: The approaches described in this thesis are able to ignore initialisation methods (denoted by the macro `module_init()`) — but neither their called functions nor similar macros. Instead of just indicating the use of a code point the FLIPPER approach can easily be expanded to count the quantity of each execution. This will create opportunities to distinguish between frequently used functions and functions only involved once. By the adoption important methods are called multiple times and the granularity is coarse-grained enough, the latter ones can be removed from the traced code point set. Of course, a categorisation is necessary to prevent the elimination of, for example, important boot functions which might also be called only once. A solution might be an automated evaluation of traces on several systems with different configuration to recognize ordinary frequency.

As an alternative, the system run time can be divided in different epochs[79], to automatically identify modules only utilized at boot time (because of the `module_init()` function). However, the removal suffers from the same problems as mentioned above and these concepts favouring a more “slimmer” kernel at the expense of accuracy and possible missing functionality.

In addition, a prioritization list guiding the SAT solver at alternative features might be offer possibilities to optimize nonfunctional properties like the power-consumption or binary size. First of all, the VAMOS UNDERTAKER needs the ability to incorporate with such feature quantifier. Afterwards, experts can evaluate each module, either by computing or just by testing. Yet, as fundamental requirements, both enough real free alternatives and sufficiently fine-grained feature granularity are necessary to make a measurable difference according to the properties.

[79] For example, the Linux runlevel can be taken into account to classify (coarse-grained) epochs.

Appendices

Appendix A

Development

A.1 Injection Examples

For the code modification I had to consider several different cases. A brief overview of the most common cases are presented below using real world examples including the injected code.

Single Statement Block

```
652 #ifdef CONFIG_X86_32
653   else if (cpu_has(c, X86_FEATURE_PAE) || cpu_has(c, ↘
        X86_FEATURE_PSE36))
  +SET_DURDEN_BIT(1801) ,
654     c->x86_phys_bits = 36;
655 #endif
```

Listing A.1 – Injection in single statement blocks without curly braces (Linux v3.15 source file `arch/x86/kernel/cpu/common.c`)

Branch Table

```
274     break;
275 #ifdef CONFIG_DEBUG_HOTPLUG_CPU0
276   case PM_RESTORE_PREPARE:
  +SET_DURDEN_BIT(2062);
      /* stripped comment */
282     if (!cpu_online(0))
283       _debug_hotplug_cpu(0, 1);
284     break;
285   case PM_POST_RESTORE:
```

```
+SET_DURDEN_BIT(2063);
    /* stripped comment */
309     _debug_hotplug_cpu(0, 0);
310   break;
311 #endif
312   default:
```

Listing A.2 – Conditional block in branch table (switch statement) with multiple branches
(Linux v3.15 source file `arch/x86/power/cpu.c` with comments removed)

Complete Expressions

```
76 if (memory_corruption_check == -1) {
77   memory_corruption_check =
78 #ifdef CONFIG_X86_BOOTPARAM_MEMORY_CORRUPTION_CHECK
+( ( SET_DURDEN_BIT(1478) ) ,
79     1
+)
80 #else
+( ( SET_DURDEN_BIT(1479) ) ,
81     0
+)
82 #endif
83     ;
84 }
```

Listing A.3 – Injection in complete expressions
(Linux v3.15 source file `arch/x86/kernel/check.c`)

Inside Expressions

```
2303   nested_vmx_entry_ctls_high &=
2304 #ifdef CONFIG_X86_64
+( ( SET_DURDEN_BIT(2322) ) , 0 ) +
2305     VM_ENTRY_IA32E_MODE |
2306 #endif
2307     VM_ENTRY_LOAD_IA32_PAT;
```

Listing A.4 – Conditional block inside expression with postfix operator
(Linux v3.15 source file `arch/x86/kvm/vmx.c`)

```
301   size = nlmsg_total_size(sizeof(struct nfgenmsg))
302        + nla_total_size(sizeof(struct nfqnl_msg_packet_hdr))
303        + nla_total_size(sizeof(u_int32_t))     /* ifindex */
304        + nla_total_size(sizeof(u_int32_t))     /* ifindex */
305 #ifdef CONFIG_BRIDGE_NETFILTER
   ++ ( ( SET_DURDEN_BIT(28603) ) , 0 )
306        + nla_total_size(sizeof(u_int32_t))     /* ifindex */
307        + nla_total_size(sizeof(u_int32_t))     /* ifindex */
308 #endif
309        + nla_total_size(sizeof(u_int32_t))     /* mark */
310        + nla_total_size(sizeof(struct nfqnl_msg_packet_hw))
311        + nla_total_size(sizeof(u_int32_t))     /* skbinfo */
312        + nla_total_size(sizeof(u_int32_t));    /* cap_len */
```

Listing A.5 – Conditional block inside expression with prefix operator
(Linux v3.15 source file `net/netfilter/nfnetlink_queue_core.c`)

A.2 Macro Defined Function

```
391 #define SHOW_FUNCTION(__FUNC, __VAR, __CONV)                \
392 static ssize_t __FUNC(struct elevator_queue *e, char *page) \
393 {                                                           \
394   struct deadline_data *dd = e->elevator_data;              \
395   int __data = __VAR;                                       \
396   if (__CONV)                                               \
397     __data = jiffies_to_msecs(__data);                      \
398   return deadline_var_show(__data, (page));                 \
399 }
```

Listing A.6 – Functions definied in macros
(Linux v3.15 source file `block/deadline-iosched.c`)

A.3 Excluded Files

Files (and directories) excluded from the injection process

- all files in */Documentation/* since there are only examples
- all files in */tools/* (compile time tools)
- all files in */scripts/* (compile time tools too)
- all files in */user/* (user space libraries)
- all files in *boot/* (sub)folders since it is code loading the kernel on early boot
- all files in *asm/* and *asm-generic/* (sub)folders (the presented approach is not able to patch assembler code)
- all files in *firmware/* (sub)folders since it contains only binary files
- the kernel module of the injection tool itself identicated by the file *durden.c* and its header
- all files in */trace/* folder and the files *ftrace.c* and *ptrace.c* (and their header files) as patching the trace infrastructure will result in a performance impact
- */include/linux/license.h* since it has no real variablilty section and is included in user space
- all files in */arch/x86/vdso/* folder since virtual dynamically linked shared objects cannot access kernel space
- */arch/arm/boot/compressed/misc.c*
- */arch/arm/kernel/process.c*
- */drivers/gpu/drm/radeon/mkregtable.c*
- */include/generated/autoconf.h*
- */include/linux/decompress/mm.h*
- */include/linux/zutil.h*
- */lib/crc32defs.h*
- */lib/decompress_bunzip2.c*
- */lib/decompress.c*
- */lib/decompress_inflate.c*
- */lib/decompress_inflate.c*
- */lib/decompress_unlzma.c*
- */lib/decompress_unlzo.c*
- */lib/decompress_unxz.c*
- */lib/gen_crc32table.c*
- */lib/raid6/mktables.c*
- */lib/zlib_inflate/inffast.c*
- */lib/zlib_inflate/inflate.c*
- */lib/zlib_inflate/inftrees.c*
- */lib/zlib_inflate/infutil.c*
- *mach/uncompress.h*

Files only excluded from preprocessor injection:

- */arch/arm/kernel/process.c*
- */drivers/base/node.c*
- */drivers/net/wan/sbni.c*
- */drivers/net/wireless/rtl8192cu/hal/rtl8192c/rtl8192c_dm.c*
- */drivers/net/wireless/rtl8192cu/os_dep/linux/ioctl_linux.c*
- */drivers/staging/comedi/drivers.c*
- */drivers/staging/prima/CORE/MAC/src/pe/lim/limProcessSmeReqMessages.c*
- */drivers/staging/prima/CORE/TL/src/wlan_qct_tl.c*
- */drivers/staging/wlags49_h2/hcf.c*
- */drivers/usb/host/dwc_otg/dwc_otg_driver.c*
- */drivers/video/msm/msm_fb.c*
- */include/linux/elfcore.h*
- */include/linux/vmstat.h*
- */lib/zlib_inflate/inffast.c*
- */net/core/net-sysfs.c*
- */net/ipv4/inet_diag.c*
- */net/ipv4/ip_gre.c*

A.4 FLIPPER in Coccinelle

In contrast to the PERL-based prototype, the final version of FLIPPER implemented in SmPL is rather compact. It makes use from the embedded PYTHON functionality and was successfully tested with the latest COCCINELLE version (1.0.0 release candidate 21 from April 13, 2014).

```
1  // Usage:
2  // spatch --sp-file this.cocci target.c --out-place -D ↘
       macro=SET_FLIPPER_BIT
3
4  virtual ignoreInitFunctions
5
6  // Redirect output of map data to seperate file
7  @ initialize:python @
8   m << virtual.mapfile;
9  @@
10 import sys
11 sys.stdout = open(m, "w")
12
13 // Initialize the blacklist, compile the blacklist regex
14 // (if enabled by the spatch flag "-D blacklist=[Blacklist-RegEx]")
15 @ script:python initBlacklist @
16 b << virtual.blacklist;
17 @@
18 global blacklistLastFileCache;
19 global blacklistRegex;
20 blacklistLastFileCache=""
21 import re
22 blacklistRegex=re.compile(b)
23
24 // Get a(ny) position
25 // in order to retrieve the file name for blacklist processing
26 @ currentFile depends on initBlacklist @
27 metavariable a;
28 position p;
29 @@
30 a@p
31
32 // Compare current file to blacklist regex
33 // (and continue with next file on match)
34 @ script:python checkBlacklist depends on currentFile @
35 p << currentFile.p;
36 @@
37 global blacklistLastFileCache
38 global blacklistRegex;
39 if blacklistLastFileCache != p[0].file:
40     if blacklistRegex.match(p[0].file):
41         cocci.exit()
```

```
42      else:
43          blacklistLastFileCache = p[0].file
44
45  // Retrieve module_init (and module_exit) functions
46  // to avoid automatic bit sets on boot
47  @ collectInitFunctions depends on ignoreInitFunctions @
48  identifier f;
49  declarer name module_init, module_exit;
50  @@
51  (
52  module_init(f);
53  |
54  module_exit(f);
55  )
56
57  // If there are some global include directives, append the flipper one
58  @ globalHeader depends on checkBlacklist @
59  @@
60  #include <...>
61  +#include <linux/flipper.h>
62
63  // If there are only some local include directives (without global ones),
64  // append the flipper include in the line above
65  @ localHeader depends on checkBlacklist && !globalHeader @
66  @@
67  +#include <linux/flipper.h>
68  #include "..."
69
70  // Insert flipper macro in every non-empty function
71  // with an increasing counter index
72  @ replace @
73  identifier f, virtual.macro;
74  fresh identifier n = "";
75  declaration d;
76  statement s,t;
77  position p;
78  @@
79  f(...) {
80  (
81  (
82  ... when != t
83      when any
84  d@p
85  +;macro(n);
86  s
87  ...
88  )
89  |
90  (
91  ... when != t
```

```
92      when any
93  d@p
94  +;macro(n);
95  )
96  |
97  (
98  +;macro(n);
99  s@p
100 ...
101 )
102 )
103 }
104
105 // Print the index, file and line number of each insertion
106 // (the index is necessary since there is an internal re-sort)
107 @ script:python print depends on replace @
108 p << replace.p;
109 n << replace.n;
110 @@
111 print "%013d\t%s:%s" % (float(n), p[0].file, p[0].line)
112
113 // Remove the (fresh inserted) flipper macros from each init function
114 // (if enabled by the spatch flag "-D ignoreInitFunctions")
115 @ cleanInitFunctions depends on collectInitFunctions && ↘
      replace @
116 identifier collectInitFunctions.f, virtual.macro;
117 @@
118 f(...){
119 ...
120 -;macro(...);
121 ...
122 }
123
124 // If there are no other include directives insert flipper
125 @ funcHeader depends on replace && !globalHeader && ↘
      !localHeader @
126 identifier replace.f;
127 @@
128 +
129 +#include <linux/flipper.h>
130 f(...){
131 ...
132 }
```

Listing A.7 – Complete SmPL source of the final FLIPPER implementation

Appendix B

Evaluation

B.1 Raspberry Pi

Whitelist

This generic whitelist consisting of 14 items is used in every Raspberry Pi scenario:

- CONFIG_AEABI
- CONFIG_ARM_ERRATA_326103
- CONFIG_ARM_ERRATA_364296
- CONFIG_ARM_ERRATA_411920
- CONFIG_BCM2708_VCHIQ
- CONFIG_DEVTMPFS_MOUNT
- CONFIG_EXT4_FS
- CONFIG_LBDAF
- CONFIG_MMC_BLOCK
- CONFIG_MMC_SDHCI_BCM2708_DMA
- CONFIG_SCSI_LOWLEVEL
- CONFIG_USB_DWCOTG
- CONFIG_VT_HW_CONSOLE_BINDING
- CONFIG_INOTIFY_USER

Hardware specification

The Raspberry Pi Model B / Rev 2 (2011.12) technical specification according to the producer [43]:

System-on-a-chip (SoC)	Broadcom BCM2835
CPU (part of SoC)	700 MHz ARM11 ARM1176JZF-S core
GPU (part of SoC)	Broadcom VideoCore IV OpenGL ES 2.0 OpenVG 1080p30 H.264 high-profile encode/decode
RAM	512 MByte
Storage	none (SD card)
USB	2 (USB 2.0)
Video output	Composite video / Composite RCA HDMI
Audio output	TRS connector | 3.5 mm jack HDMI
Low-level peripherals	General Purpose Input/Output (GPIO) pins Serial Peripheral Interface Bus (SPI) I^2C I^2S Universal asynchronous receiver/transmitter (UART)
Network	10/100 wired Ethernet RJ45
Power	700 mA (3.5 W)

Coder

Summary: My test case used a Coder v0.4 image employing a Linux kernel v3.6.11 while the Raspberry Pi was only connected with wired ethernet.
Using FLIPPER, the tracing lasted 1182 s, but the configuration was stable instantly after finishing boot with 348 enabled features and 81 modules.
With FTRACE the tracing lasted 1227 s, the configuration was stable 1200 s with 316 enabled features and 55 modules.

time	action
0 s	*system boot* idling
600 s	connecting via web browser modifying and testing "Space Rocks!"
1200 s	*shut down (triggered automatically)*

Table B.1 – Detailed schedule for the Coder scenario on Raspberry Pi

Metric	Baseline	Tailored using FLIPPER	Tailored using FTRACE
static features	640	348 (54.38%)	316 (49.38%)
modules	1,038	81 (7.80%)	55 (5.30%)
value features	54	44 (81.85%)	34 (62.96%)
configuration items	1,732	470 (27.14%)	405 (23.38%)
text size (bytes)	22,621,072	4,835,648 (21.38%)	3,484,020 (15.40%)
data size (bytes)	1,437,002	223,132 (15.53%)	182,632 (12.71%)
bss size (bytes)	2,237,125	683,504 (30.55%)	352,656 (15.76%)
compiled C lines	845,627	239,680 (28.34%)	170,114 (20.12%)
compiled C files	4,166	1,465 (35.17%)	1,115 (26.76%)

Table B.2 – Detailed kernel comparison for tailoring of Coder

Statistics for tailoring Coder using FLIPPER

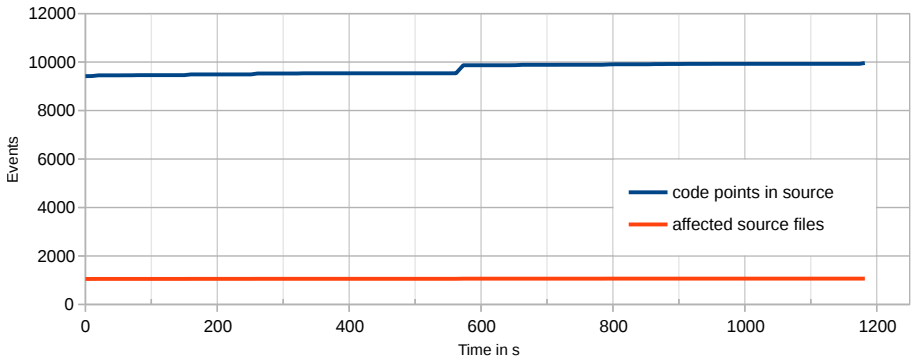


Figure B.1 – Traced events per time during evaluation of Coder using FLIPPER

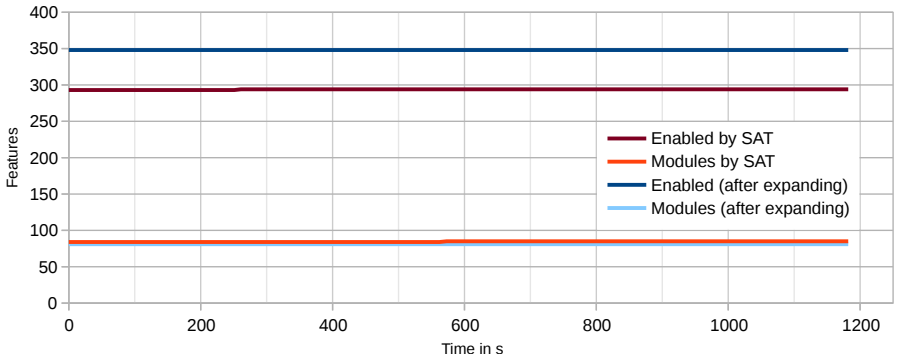


Figure B.2 – Evolution of features during evaluation of Coder using FLIPPER

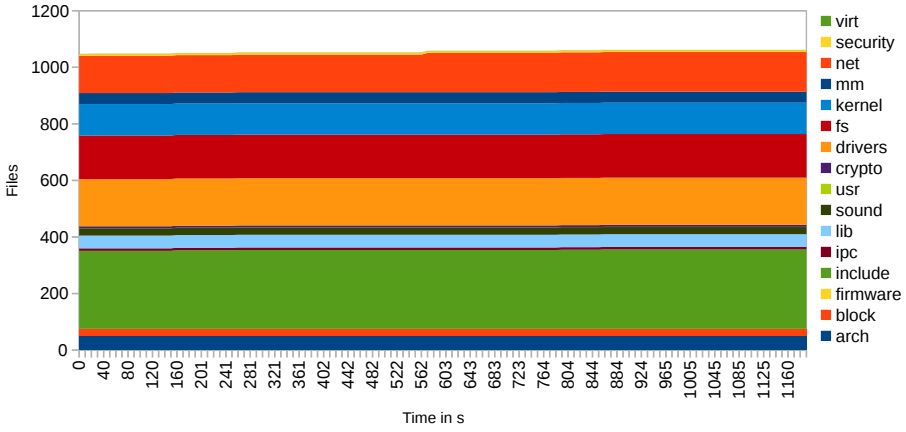


Figure B.3 – Traced events per directory and time during evaluation of Coder using FLIPPER

Statistics for tailoring Coder using FTRACE

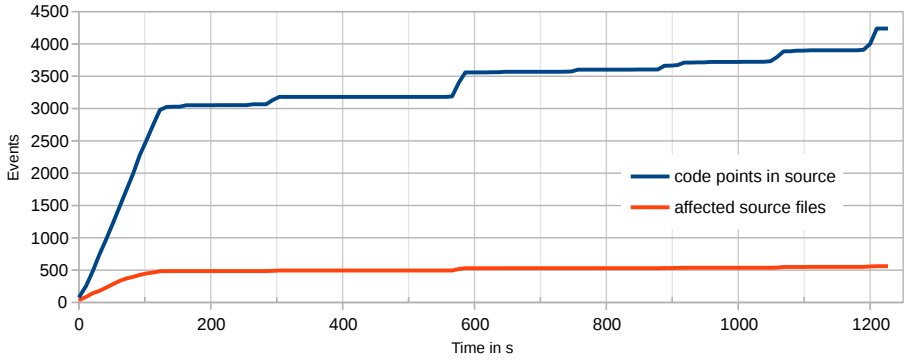


Figure B.4 – Traced events per time during evaluation of Coder using FTRACE

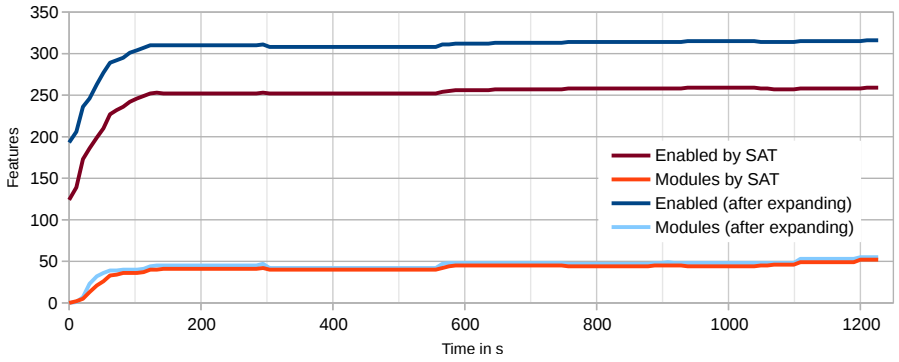


Figure B.5 – Evolution of features during evaluation of Coder using FTRACE

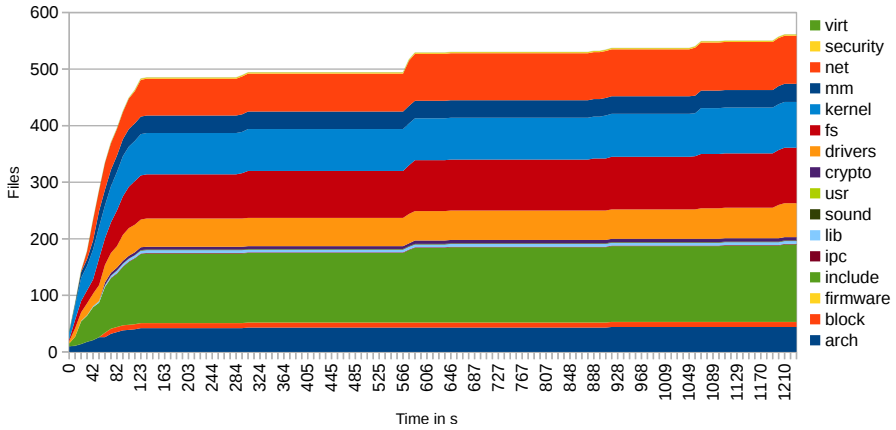


Figure B.6 – Traced events per directory and time during evaluation of Coder using FTRACE

OnionPi

Summary: This test case used a raspbian based image employing a Linux kernel v3.6.11 with OnionPi applications installed (set up at September 27th, 2013), while the Raspberry Pi was connected to wired ethernet and a USB WiFi stick.
Using FLIPPER, the tracing lasted 1196 s, but the configuration was stable instantly after finishing boot with 349 enabled features and 77 modules.
With FTRACE the tracing lasted 1226 s, the configuration was stable after 727 s with 287 enabled features and 33 modules.

time	action
0 s	*system boot* idling
600 s	connection to wlan access point using laptop browsing websites (http and https) retrieving mails
900 s	connecting via mobile phone browsing websites
1200 s	*shut down (trigged automatically)*

Table B.3 – Detailed schedule for the OnionPi scenario on Raspberry Pi

Metric	Baseline	Tailored using FLIPPER	Tailored using FTRACE
static features	640	349 (54.53%)	287 (44.84%)
modules	1,038	77 (7.42%)	33 (3.18%)
value features	54	43 (79.63%)	34 (62.96%)
configuration items	1,732	469 (27.08%)	354 (20.44%)
text size (bytes)	22,688,201	5,041,604 (22.22%)	3,972,552 (17.51%)
data size (bytes)	1,437,062	310,240 (21.59%)	275,828 (19.19%)
bss size (bytes)	2,237,221	698,924 (31.24%)	368,768 (16.48%)
compiled C lines	846,554	252,362 (29.81%)	197,512 (23.33%)
compiled C files	4,167	1,490 (35.76%)	1,189 (28.53%)

Table B.4 – Detailed kernel comparison for tailoring of OnionPi

Statistics for tailoring OnionPi using FLIPPER

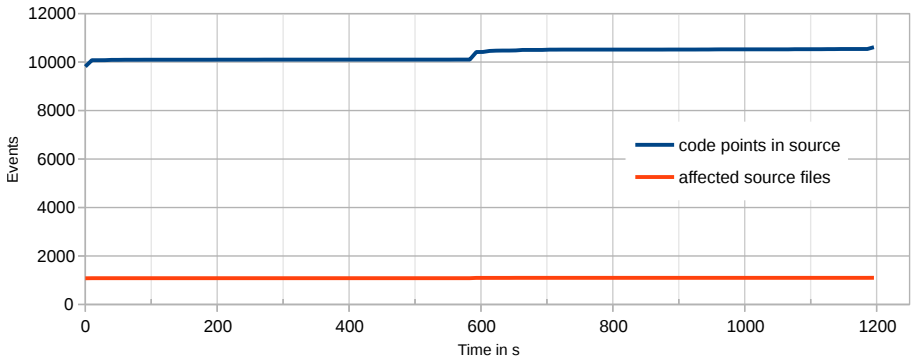


Figure B.7 – Traced events per time during evaluation of OnionPi using FLIPPER

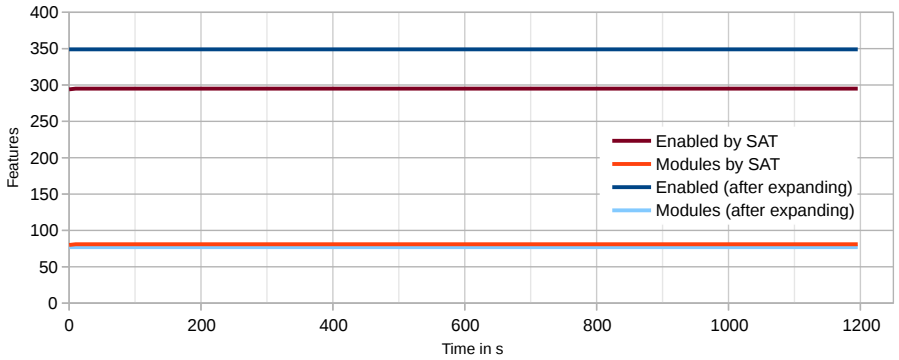


Figure B.8 – Evolution of features during evaluation of OnionPi using FLIPPER

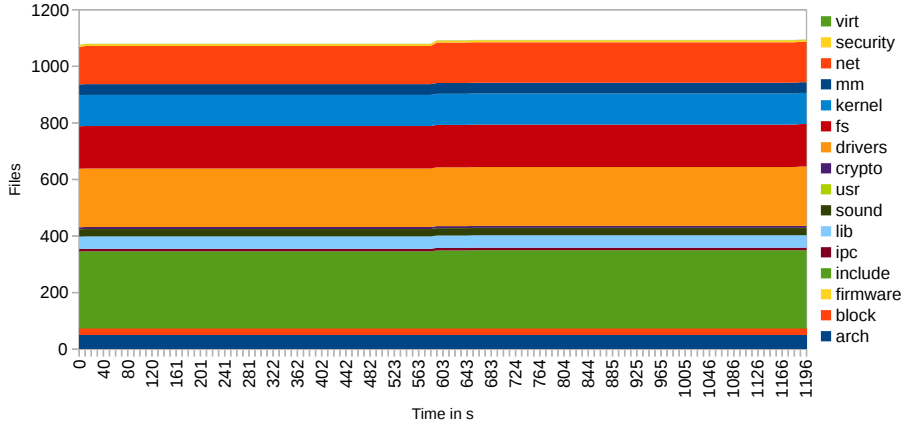


Figure B.9 – Traced events per directory and time during evaluation of OnionPi using FLIPPER

Statistics for tailoring OnionPi using FTRACE

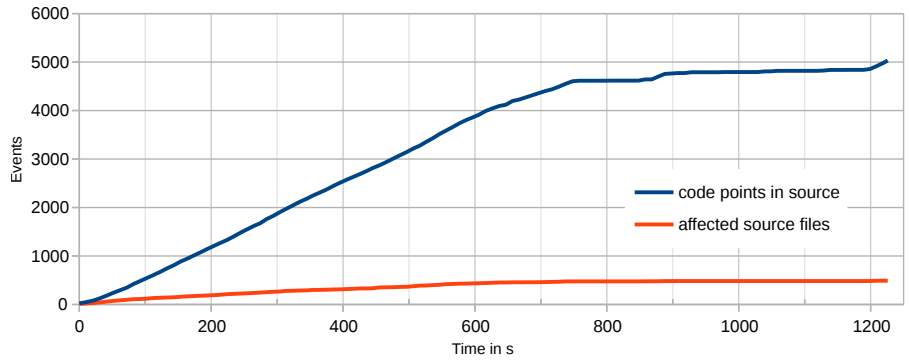


Figure B.10 – Traced events per time during evaluation of OnionPi using FTRACE

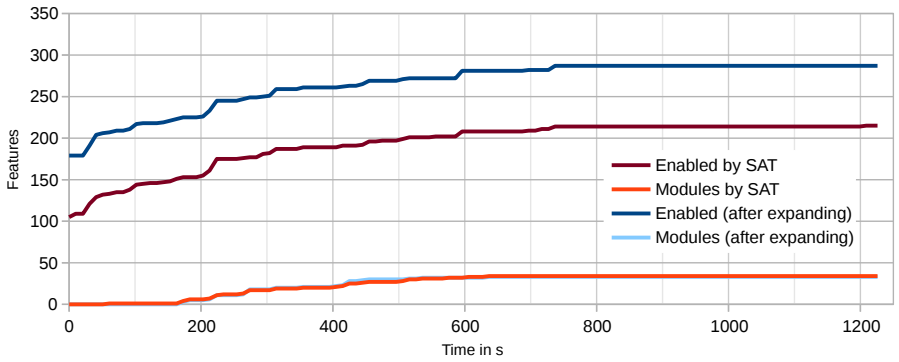


Figure B.11 – Evolution of features during evaluation of OnionPi using FTRACE

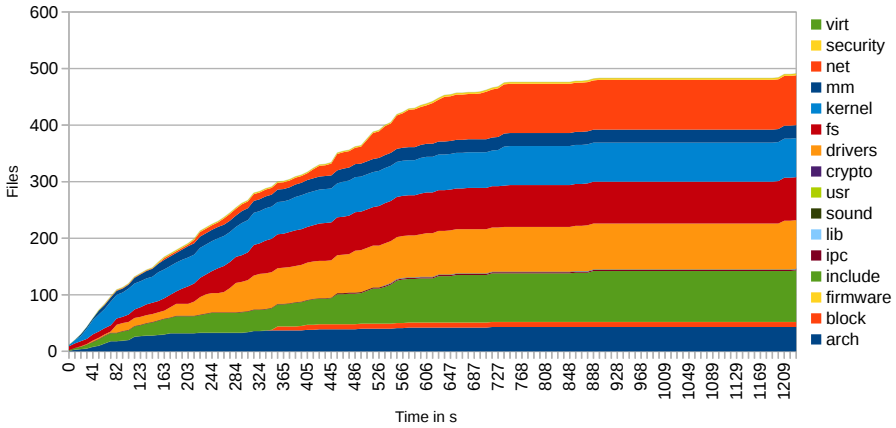


Figure B.12 – Traced events per directory and time during evaluation of OnionPi using FTRACE

raspBMC

Summary: This test case used a Raspberry Pi image (from January 1th, 2014 including the "december update") employing a Linux kernel v3.10.25, while the Raspberry Pi was connected to wired ethernet, a head phone and a monitor (using HDMI), having the hardware video decoding enabled.
Using FLIPPER, the tracing lasted 1196 s, but the configuration was stable after only 20 s (after finishing boot up) with 352 enabled features and 100 modules.
With FTRACE the tracing lasted 1218 s, the configuration was stable after 1188 s with 285 enabled features and 45 modules.

time	action
0 s	*system boot* idling
600 s	starting weather app
630 s	starting video (in MPEG format) located on external device using SFTP controlling the playback via keyboard media keys
720 s	non-privileged remote access via SSH, running TOP
780 s	remotely accessing web-based front end controlling the playback via web front end
870 s	switching to video (in MPEG2 format)
1020 s	switching to video (in MPEG4 format)
1200 s	*shut down (trigged automatically)*

Table B.5 – Detailed schedule for the raspBMC scenario

Metric	Baseline	Tailored using FLIPPER	Tailored using FTRACE
static features	663	352 (53.09%)	285 (42.84%)
modules	1,156	100 (8.65%)	45 (3.89%)
value features	55	45 (81.82%)	34 (61.82%)
configuration items	1,874	497 (26.91%)	364 (19.71%)
text size (bytes)	22,960,278	5,656,040 (24.63%)	4,458,236 (19.42%)
data size (bytes)	1,437,062	290,716 (18.98%)	268,132 (17.51%)
bss size (bytes)	707,155	351,352 (49.69%)	335,152 (47.39%)
compiled C lines	842,460	275,403 (32.69%)	216,941 (25.75%)
compiled C files	4,223	1,588 (37.60%)	1,301 (30.81%)

Table B.6 – Detailed kernel comparison for tailoring of raspBMC

Feature comparison

By comparing the features of the baseline with the tailored ones, you can observe 13 items not listed in the baseline. Many of them belong to debug purposes.

- CONFIG_ARM_PATCH_PHYS_VIRT
- CONFIG_COREDUMP
- CONFIG_CPU_FREQ_DEFAULT_GOV_PERFORMANCE
- CONFIG_DEBUG_BUGVERBOSE
- CONFIG_DEBUG_MEMORY_INIT
- CONFIG_DEFAULT_NOOP
- CONFIG_DEFAULT_RENO
- CONFIG_ELF_CORE
- CONFIG_KALLSYMS
- CONFIG_NAMESPACES
- CONFIG_UID16
- CONFIG_UIDGID_CONVERTED
- CONFIG_VM_EVENT_COUNTERS

The eight items only present in the FLIPPER are solely used for different decompression of the initial ram disk:

- CONFIG_DECOMPRESS_BZIP2
- CONFIG_DECOMPRESS_LZMA
- CONFIG_DECOMPRESS_LZO
- CONFIG_DECOMPRESS_XZ
- CONFIG_RD_BZIP2
- CONFIG_RD_LZMA
- CONFIG_RD_LZO
- CONFIG_RD_XZ

The item only present in the FTRACE approach is a memory allocation debugging routine

- CONFIG_DEBUG_SLAB

Statistics for tailoring raspBMC using FLIPPER

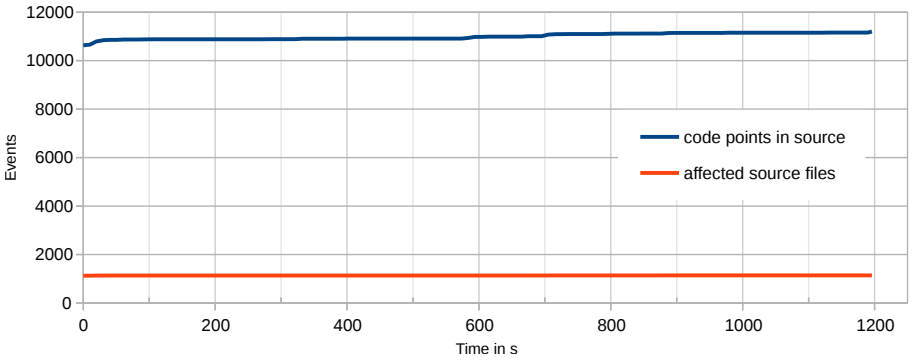


Figure B.13 – Traced events per time during evaluation of raspBMC using FLIPPER

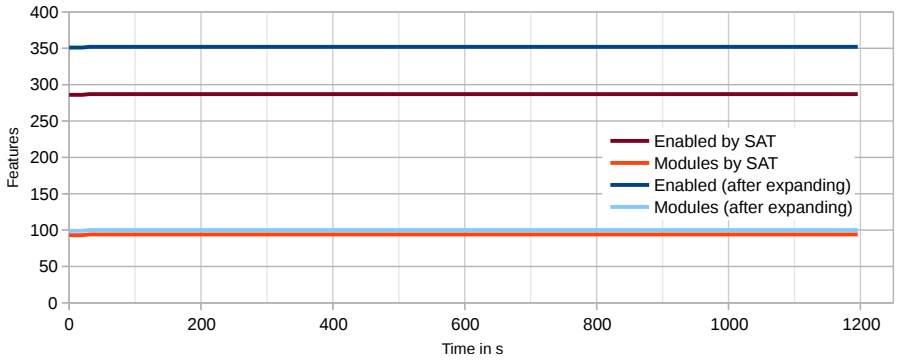


Figure B.14 – Evolution of features during evaluation of raspBMC using FLIPPER

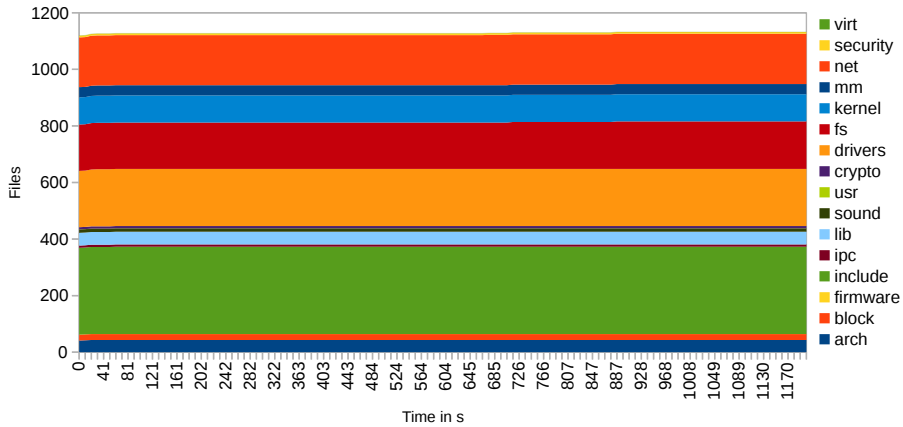


Figure B.15 – Traced events per directory and time during evaluation of raspBMC using FLIPPER

Statistics for tailoring raspBMC using FTRACE

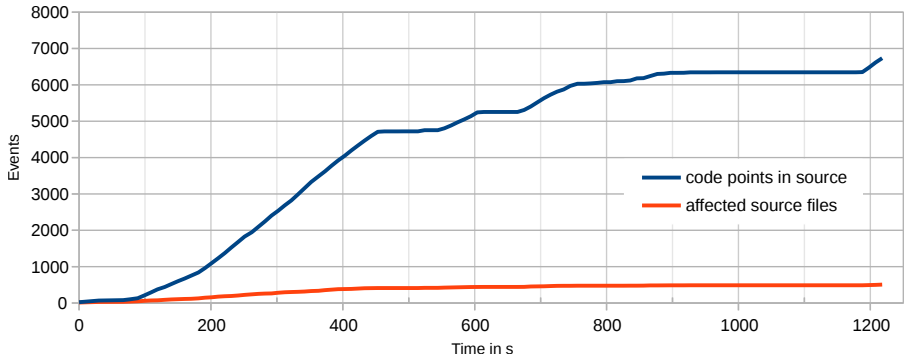


Figure B.16 – Traced events per time during evaluation of raspBMC using FTRACE

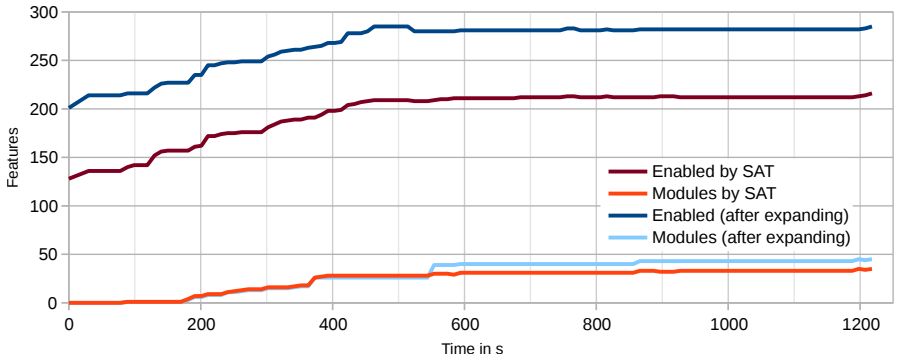


Figure B.17 – Evolution of features during evaluation of raspBMC using FTRACE

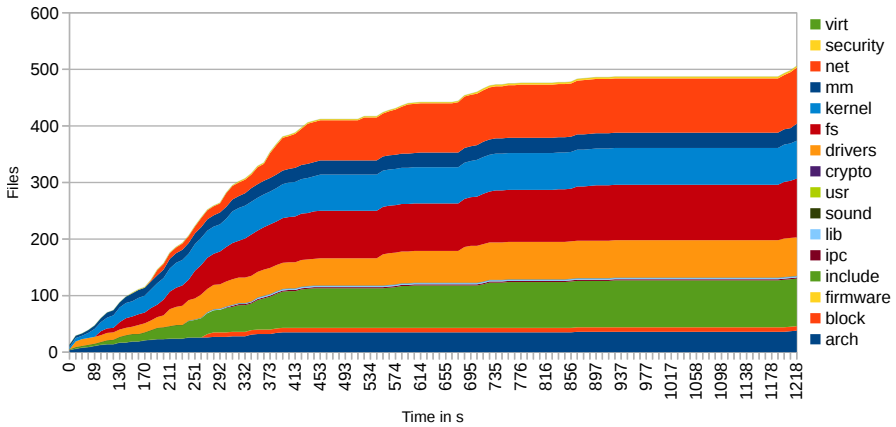


Figure B.18 – Traced events per directory and time during evaluation of raspBMC using FTRACE

B.2 Google Nexus 4

Whitelist

This whitelist consisting of 14 items is used in the Google Nexus 4 scenario:

- CONFIG_BUG
- CONFIG_DIAG_OVER_USB
- CONFIG_EMBEDDED
- CONFIG_FB_MSM_MIPI_LGIT_VIDEO_WXGA_PT_PANEL
- CONFIG_KERNEL_MSM_CONTIG_MEM_REGION
- CONFIG_MMC_MSM_SDC1_SUPPORT
- CONFIG_MSM_CSI20_HEADER
- CONFIG_MSM_N_WAY_SMSM
- CONFIG_NEON
- CONFIG_RD_GZIP
- CONFIG_TOUCH_REG_MAP_TM2000
- CONFIG_USB_EHCI_MSM
- CONFIG_VIDEO_V4L2_SUBDEV_API
- CONFIG_WCD9310_CODEC

Hardware specification

The LG E960 / Google Nexus 4 technical specification according to the producer [30]:

CPU	Qualcomm Snapdragon S4 Pro 1.5 GHz
RAM	2 GByte
Storage	8 GByte
Screen	4.7” 1280×768 (320ppi)
Camera	8 MP (main) 1.3 MP (front)
Sensors	Accelerometer Compass Ambient light Proximity Gyroscope Pressure GPS
Network	GSM/EDGE/GPRS (850, 900, 1800, 1900 MHz) 3G (850, 900, 1700, 1900, 2100 MHz) HSPA+
Wireless	Wi-Fi (802.11 b/g/n) SlimPort NFC (Android Beam) Bluetooth
Battery	2100 mA h

Ubuntu Touch

Summary: I performed the trace on a Ubuntu Touch Saucy (release of November 27, 2013) with Linux kernel 3.4.0, the total tracing time was 1221 s while the configuration was stable after 877 s with 750 enabled features.

time	action
0 s	*system boot* idling
600 s	activating smartphone (from standby) navigating through menu
690 s	activating wireless LAN connecting to access point
720 s	outgoing phone call
780 s	starting camera app taking pictures with front and main camera (involving flash light)
840 s	using web browser
900 s	incoming call (without answering it)
1020 s	connecting with PC using USB remote file access
1080 s	shell access via USB
1110 s	setting display brightness
1140 s	switching to standby
1200 s	*shut down (trigged automatically)*

Table B.7 – Detailed schedule for tracing Ubuntu Touch on Google Nexus 4

	Metric	Baseline	Tailored (FLIPPER)	
features	static	974	752	(77.21%)
	modules	145	0	(0.00%)
	values	67	59	(88.06%)
configuration items		1,186	811	(68.38%)
vmlinux	text size	13,489,768 bytes	12,037,224 bytes	(89.23%)
	data size	1,206,812 bytes	1,171,756 bytes	(97.10%)
	bss size	2,735,516 bytes	2,587,100 bytes	(94.57%)
	compiled C code	542,874 lines	494,082 lines	(91.01%)
	taken from	2,497 files	2,233 files	(89.26%)
total	text size	14,464,220 bytes	12,037,224 bytes	(83.22%)
	data size	1,312,683 bytes	1,171,756 bytes	(89.26%)
	bss size	2,745,447 bytes	2,587,100 bytes	(94.23%)
	compiled C lines	564,324 lines	494,082 lines	(87.55%)
	taken from	2,776 files	2,233 files	(80.44%)

Table B.8 – Detailed kernel comparison for Ubuntu Touch

Statistics for tailoring Ubuntu Touch (using FLIPPER)

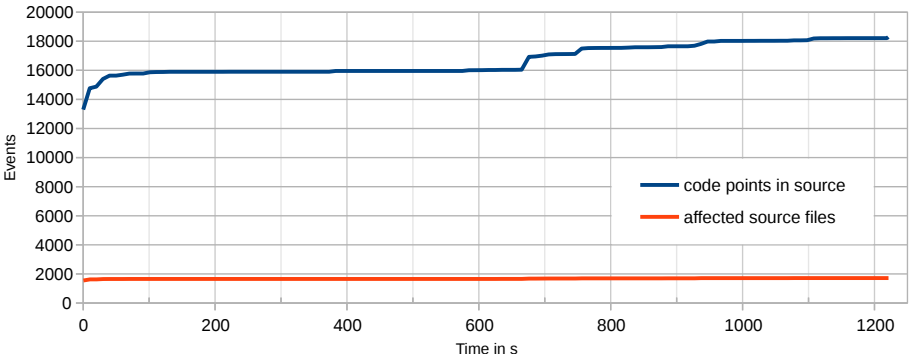


Figure B.19 – Traced events per time during evaluation of Ubuntu Touch (using FLIPPER)

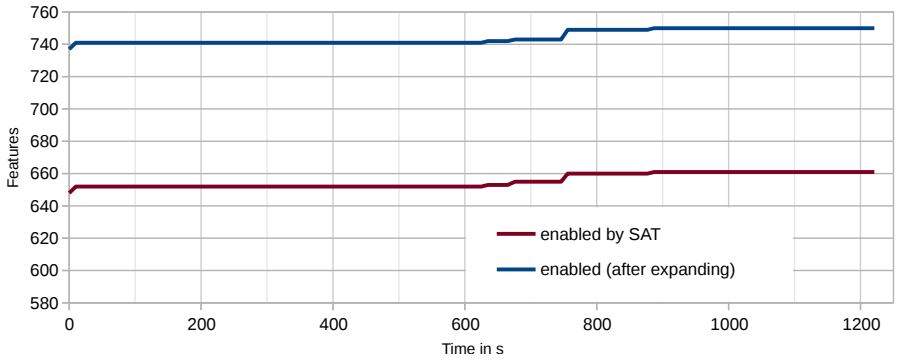


Figure B.20 – Evolution of features during evaluation of Ubuntu Touch (using FLIPPER)

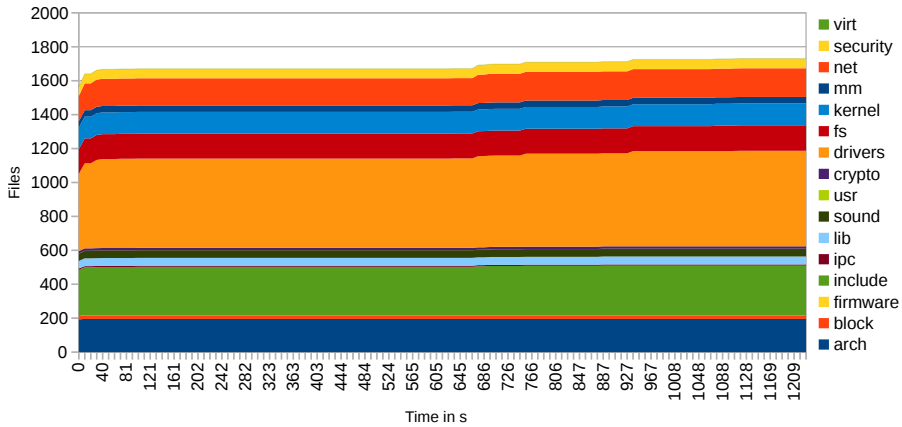


Figure B.21 – Traced events per directory and time during evaluation of Ubuntu Touch (using FLIPPER)

B.3 Emulation

Qemu Coccinelle

The semantic patch used to enable the tracking of the virtual machines program counter:

```
@@
declaration d;
statement s,t;
@@

tb_gen_code(...) {
... when != t
  when any
d
+printf("%016llX\n", (unsigned long long int) pc);
s
...
}
```

Listing B.1 – SmPL patch for QEMU enabling output of all block starting addresses

Workflow

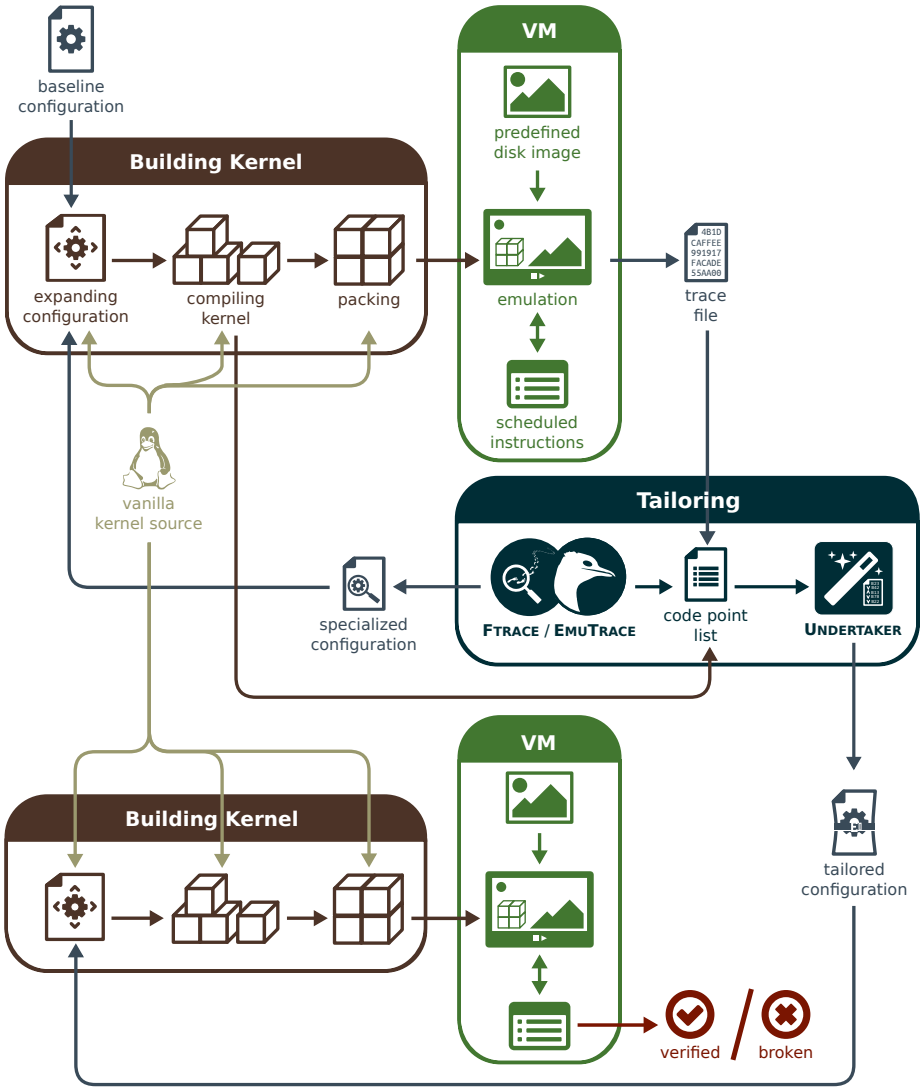


Figure B.22 – Schematic representation for the emulation workflow of both the traditional FTRACE based approach and the Emulator-based Code-Point Recording

Whitelist

Method	Kernel	CONFIG_BINFMT_SCRIPT	CONFIG_DEVTMPFS_MOUNT	CONFIG_INOTIFY_USER	CONFIG_OPTPROBES	CONFIG_RD_GZIP	CONFIG_UNIX	∑
DURDEN and FLIPPER	v3.10	✓					✓	2
	v3.11	✓					✓	2
	v3.12	✓					✓	2
	v3.13	✓					✓	2
	v3.14	✓					✓	2
	v3.15	✓				✓	✓	3
FTRACE and FTRACE (EARLY BOOT)	v3.10	✓	✓	✓		✓	✓	5
	v3.11	✓	✓			✓	✓	4
	v3.12	✓	✓	✓		✓	✓	5
	v3.13	✓	✓	✓		✓	✓	5
	v3.14	✓	✓	✓		✓	✓	5
	v3.15	✓	✓	✓		✓	✓	5
EMUTRACE	v3.10	✓					✓	2
	v3.11	✓					✓	2
	v3.12	✓					✓	2
	v3.13	✓					✓	2
	v3.14	✓					✓	2
	v3.15	✓				✓	✓	3
EMUTRACE (NO INIT)	v3.10	✓					✓	2
	v3.11	✓					✓	2
	v3.12	✓					✓	2
	v3.13	✓			✓		✓	3
	v3.14	✓			✓		✓	3
	v3.15	✓			✓	✓	✓	4

Table B.9 – Necessary whitelist items for the emulation based evaluation

Schedule

Although this schedule is the common one, I had to consider use case depended enhancements: The full emulation is significant slower than the one supported by the kernel virtual machine. Therefore, I had to increase the idle time (about 3000 s). Anyway, besides this differences in timing, there is no change in the executed actions nor their order.

time	action
0 s	*system boot* idling
600 s	*initial test:* testing SSH connection fetching static web page via HTTP fetching static web page via HTTPS fetching dynamic (PHP) web page via HTTP
660 s	*testing remote access:* connecting via SSH executing file system commands file transfer via SCP
690 s	*testing web server:* fetching bigger sized static web page[80] via HTTPS
	calculating (and verifying) prime numbers in dynamic web page via HTTP 1337th prime number 1111th prime number flushing PHP OPCACHE[81] 1500th prime number (suitable on most systems with 1 GByte memory) 1800th prime number (suitable on systems with 1 GByte free memory) 2222th prime number (fails on systems with 1 GByte memory) 3000th prime number (suitable on most systems with 4 GByte memory) 3100th prime number (suitable on systems with 1 GByte free memory) 3333th prime number (fails on systems with 1 GByte memory) 27070th prime number (OUT-OF-MEMORY KILLER[82] will abort calculation) concurrent calculation of 1040th - 1050th prime number concurrent calculation of 1549th - 1555th prime number
	transferring trace files *shut down*

Table B.10 – Detailed schedule for the automatic simulation actions in the emulation approach

[80] I decided to use the JAVASCRIPT based open source browser game "2048" [11] as a real world example.

[81] OPcache stores precompiled PHP scripts in memory — for more information take a view at the official documentation [38].

[82] Since the environment is based on DEBIAN JESSIE, which implies strict memory management rules in its default configuration, the simulation takes account of disruptions caused by the OOM KILLER [37].

Linux kernel	Method	collected addresses	code points	**traceable** static	**traceable** module	**partial** static	**partial** module	**expanded** features static	**expanded** features module	**expanded** features summarized
v3.15	Baseline							1,402	2,741	4,143
	DURDEN	13,860	13,860	1,409	2,737	579	96	674 (48.07%)	97 (3.54%)	771 (18.61%)
	FLIPPER	13,064	13,064	1,408	2,737	579	97	673 (48.00%)	96 (3.50%)	769 (18.56%)
	FTRACE	7,731	7,009	1,413	2,737	322	15	421 (30.03%)	17 (0.62%)	438 (10.57%)
	FTRACE (EARLY BOOT)	7,388	6,767	1,412	2,737	331	15	435 (31.03%)	17 (0.62%)	452 (10.91%)
	EMUTRACE	14,128,732	84,577	4,026	0	847	2,431	988 (70.47%)	2,502 (91.28%)	3,490 (84.24%)
	EMUTRACE (NO INIT)	20,284,134	76,962	4,026	0	798	1,618	931 (66.41%)	1,706 (62.24%)	2,637 (63.65%)
v3.14	Baseline							1,400	2,750	4,150
	DURDEN	13,786	13,786	1,407	2,746	574	96	698 (49.86%)	96 (3.49%)	794 (19.13%)
	FLIPPER	12,981	12,981	1,406	2,746	574	97	696 (49.71%)	95 (3.45%)	791 (19.06%)
	FTRACE	7,053	6,510	1,411	2,746	320	15	462 (33.00%)	17 (0.62%)	479 (11.54%)
	FTRACE (EARLY BOOT)	7,747	7,022	1,410	2,746	329	15	477 (34.07%)	17 (0.62%)	494 (11.90%)
	EMUTRACE	14,269,350	84,267	4,031	0	839	2,440	1,036 (74.00%)	2,514 (91.42%)	3,550 (85.54%)
	EMUTRACE (NO INIT)	15,341,088	76,675	4,031	0	791	1,622	985 (70.36%)	1,709 (62.15%)	2,694 (64.92%)
v3.13	Baseline							1,408	2,734	4,142
	DURDEN	13,587	13,587	1,415	2,730	564	95	688 (48.86%)	95 (3.47%)	783 (18.90%)
	FLIPPER	12,676	12,676	1,414	2,730	563	96	686 (48.72%)	94 (3.44%)	780 (18.83%)
	FTRACE	7,146	6,636	1,419	2,730	325	17	472 (33.52%)	19 (0.69%)	491 (11.85%)
	FTRACE (EARLY BOOT)	7,645	7,054	1,418	2,730	318	17	458 (32.53%)	19 (0.69%)	477 (11.52%)
	EMUTRACE	10,594,175	84,058	4,025	0	829	2,432	1,030 (73.15%)	2,502 (91.51%)	3,532 (85.27%)
	EMUTRACE (NO INIT)	10,688,173	76,294	4,025	0	787	1,609	980 (69.60%)	1,697 (62.07%)	2,677 (64.63%)

Table B.11 – Collected data and feature overview by automated tailoring with different approaches for Linux kernel v3.13 – v3.15

Linux kernel	Method	collected addresses	code points	**traceable** static	**traceable** module	**partial** static	**partial** module	**expanded** features static	**expanded** features module	**expanded** features summarized
v3.12	Baseline							1,395	2,706	4,101
	DURDEN	13,507	13,507	1,402	2,702	558	94	689 (49.39%)	90 (3.33%)	779 (19.00%)
	FLIPPER	12,571	12,571	1,401	2,702	558	95	688 (49.32%)	90 (3.33%)	778 (18.97%)
	FTRACE	7,220	6,707	1,406	2,702	318	19	465 (33.33%)	21 (0.78%)	486 (11.85%)
	FTRACE (EARLY BOOT)	8,013	7,237	1,405	2,702	325	19	479 (34.34%)	21 (0.78%)	500 (12.19%)
	EMUTRACE	12,762,147	83,982	3,984	0	822	2,402	1,032 (73.98%)	2,464 (91.06%)	3,496 (85.25%)
	EMUTRACE (NO INIT)	12,568,803	76,635	3,984	0	781	1,587	979 (70.18%)	1,660 (61.35%)	2,639 (64.35%)
v3.11	Baseline							1,385	2,688	4,073
	DURDEN	13,516	13,516	1,392	2,684	559	93	692 (49.96%)	92 (3.42%)	784 (19.25%)
	FLIPPER	12,593	12,593	1,391	2,684	560	94	692 (49.96%)	92 (3.42%)	784 (19.25%)
	FTRACE	7,378	6,843	1,396	2,684	327	19	480 (34.66%)	21 (0.78%)	501 (12.30%)
	FTRACE (EARLY BOOT)	8,432	7,602	1,395	2,684	327	19	480 (34.66%)	21 (0.78%)	501 (12.30%)
	EMUTRACE	11,111,749	83,150	3,959	0	818	2,384	1,016 (73.36%)	2,459 (91.48%)	3,475 (85.32%)
	EMUTRACE (NO INIT)	11,035,048	75,422	3,959	0	772	1,568	969 (69.96%)	1,646 (61.24%)	2,615 (64.20%)
v3.10	Baseline							1,364	2,672	4,036
	DURDEN	13,255	13,255	1,371	2,668	554	95	684 (50.15%)	92 (3.44%)	776 (19.23%)
	FLIPPER	12,319	12,319	1,370	2,668	555	96	685 (50.22%)	92 (3.44%)	777 (19.25%)
	FTRACE	7,320	6,732	1,375	2,668	316	20	465 (34.09%)	24 (0.90%)	489 (12.12%)
	FTRACE (EARLY BOOT)	8,776	7,837	1,374	2,668	321	20	468 (34.31%)	24 (0.90%)	492 (12.19%)
	EMUTRACE	12,698,749	82,734	3,918	0	809	2,368	1,005 (73.68%)	2,439 (91.28%)	3,444 (85.33%)
	EMUTRACE (NO INIT)	12,068,974	75,090	3,918	0	759	1,528	949 (69.57%)	1,607 (60.14%)	2,556 (63.33%)

Table B.12 – Collected data and feature overview by automated tailoring with different approaches for Linux kernel v3.10 – v3.12

Linux kernel	Method	vmlinux only: size (in bytes) text	data	bss	vmlinux only: compiled source lines	files	.ko files	total (including LKM): size (in bytes) text	data	bss	total (including LKM): compiled source lines	files
v3.15	Baseline	7,933,254	1,374,880	5,308,416	307,799	2,049	2,943	73,499,239	1,374,880	13,731,311	4,797,533	10,462
	FLIPPER	6,553,693	1,225,888	5,169,152	275,726	1,886	98	9,866,461	1,225,888	5,563,907	394,333	2,361
	DURDEN	6,533,987	1,233,376	5,173,248	275,220	1,886	99	9,865,887	1,233,376	5,558,491	394,160	2,357
	FTRACE	4,434,753	1,077,376	765,952	187,009	1,292	18	5,574,146	1,077,376	902,987	233,056	1,492
	FTRACE (EARLY BOOT)	4,537,154	1,099,544	765,952	190,987	1,309	18	5,678,067	1,099,544	903,019	237,098	1,509
	EMUTRACE	7,247,248	1,331,832	5,479,608	312,358	2,108	2,695	53,297,546	1,331,832	11,596,264	2,226,537	9,552
	EMUTRACE (NO INIT)	7,409,902	1,323,720	5,193,728	323,331	2,121	1,885	41,454,897	1,323,720	10,326,714	1,735,278	7,627
v3.14	Baseline	7,917,430	1,343,360	5,320,704	305,615	2,036	2,953	73,354,378	1,343,360	13,800,698	2,483,439	10,436
	DURDEN	6,648,667	1,224,384	5,308,416	276,306	1,894	98	10,058,876	1,224,384	5,688,115	395,394	2,357
	FLIPPER	6,667,661	1,218,976	5,308,416	276,795	1,893	97	10,061,438	1,218,976	5,698,099	395,540	2,360
	FTRACE	4,743,367	1,093,856	983,040	193,901	1,322	18	5,941,261	1,093,856	1,115,807	240,007	1,522
	FTRACE (EARLY BOOT)	4,873,603	1,108,760	983,040	198,102	1,341	18	6,073,033	1,108,760	1,115,839	244,272	1,541
	EMUTRACE	7,357,407	1,315,640	5,320,704	311,951	2,082	2,706	54,080,347	1,315,640	11,843,747	2,230,101	9,543
	EMUTRACE (NO INIT)	7,770,706	1,331,488	5,328,896	328,740	2,135	1,888	42,184,787	1,331,488	10,471,971	1,740,078	7,614
v3.13	Baseline	7,799,502	1,331,552	5,312,512	301,583	2,011	2,950	72,916,550	1,331,552	13,781,827	2,470,599	10,389
	DURDEN	6,259,465	1,195,920	5,279,744	260,522	1,826	99	9,857,155	1,195,920	5,689,160	389,529	2,320
	FLIPPER	6,271,942	1,190,768	5,279,744	260,967	1,824	98	9,853,054	1,190,768	5,698,400	389,637	2,323
	FTRACE	4,530,856	1,078,984	958,464	184,267	1,286	20	5,925,231	1,078,984	1,124,975	240,926	1,521
	FTRACE (EARLY BOOT)	4,426,758	1,068,176	958,464	180,846	1,270	20	5,818,431	1,068,176	1,124,943	237,376	1,505
	EMUTRACE	6,978,926	1,288,696	5,292,032	296,508	2,010	2,689	53,418,578	1,288,696	11,830,599	2,209,814	9,481
	EMUTRACE (NO INIT)	7,442,181	1,305,096	5,320,704	316,724	2,071	1,878	42,135,649	1,305,096	10,515,458	1,743,583	7,630

Table B.13 – Tailored kernel binary statistics for Linux kernel v3.13 – v3.15

Linux kernel	Method	vmlinux only: size (in bytes) text	data	bss	vmlinux only: compiled source lines	files	.ko files	total (including LKM): size (in bytes) text	data	bss	total (including LKM): compiled source lines	files
v3.12	Baseline	7,672,934	1,313,360	5,308,416	298,314	1,985	2,920	72,005,558	1,313,360	13,739,344	2,445,947	10,259
	DURDEN	6,228,968	1,217,240	5,275,648	260,138	1,810	94	9,701,205	1,217,240	5,685,578	386,514	2,297
	FLIPPER	6,241,845	1,210,648	5,275,648	260,585	1,808	94	9,719,094	1,210,648	5,696,039	387,104	2,301
	FTRACE	4,445,628	1,085,952	954,368	181,487	1,269	22	5,854,328	1,085,952	1,120,005	239,098	1,505
	FTRACE (EARLY BOOT)	4,573,195	1,102,424	954,368	185,632	1,288	22	5,984,469	1,102,424	1,120,037	243,371	1,524
	EMUTRACE	7,532,593	1,352,120	5,292,032	314,409	2,053	2,657	52,807,768	1,352,120	11,725,319	2,187,917	9,359
	EMUTRACE (NO INIT)	7,711,117	1,337,168	5,300,224	329,368	2,130	1,842	41,580,988	1,337,168	10,441,448	1,721,915	7,521
v3.11	Baseline	7,560,743	1,300,368	5,304,320	294,614	1,970	2,888	67,057,147	1,300,368	12,617,811	2,328,121	9,836
	DURDEN	6,175,521	1,208,008	5,271,552	258,634	1,813	96	9,412,660	1,208,008	5,647,080	383,354	2,293
	FLIPPER	6,188,638	1,201,416	5,271,552	259,086	1,811	96	9,430,678	1,201,416	5,657,701	383,935	2,297
	FTRACE	4,522,970	1,089,104	954,368	183,680	1,283	22	5,889,301	1,089,104	1,105,420	240,773	1,516
	FTRACE (EARLY BOOT)	4,522,970	1,089,104	954,368	183,680	1,283	22	5,889,301	1,089,104	1,105,420	240,773	1,516
	EMUTRACE	6,831,275	1,293,544	5,275,648	290,743	1,997	2,654	50,984,691	1,293,544	11,679,420	2,126,436	9,185
	EMUTRACE (NO INIT)	8,086,472	1,328,160	5,304,320	344,705	2,111	1,827	39,898,797	1,328,160	10,321,344	1,663,133	7,354
v3.10	Baseline	7,447,598	1,283,088	5,292,032	291,644	1,938	2,872	66,293,115	1,283,088	12,499,443	2,296,457	9,686
	DURDEN	6,072,617	1,191,008	5,246,976	255,427	1,782	97	9,315,870	1,191,008	5,633,645	380,660	2,268
	FLIPPER	6,084,851	1,184,968	5,246,976	255,969	1,781	97	9,331,230	1,184,968	5,644,218	381,226	2,273
	FTRACE	4,501,794	1,102,152	5,144,576	182,862	1,285	25	5,867,784	1,102,152	5,295,880	240,281	1,522
	FTRACE (EARLY BOOT)	4,502,644	1,100,616	5,144,576	182,935	1,285	25	5,869,578	1,100,616	5,296,008	240,380	1,522
	EMUTRACE	6,712,221	1,281,272	5,259,264	286,457	1,956	2,633	50,250,932	1,281,272	11,546,651	2,093,593	9,051
	EMUTRACE (NO INIT)	7,200,251	1,294,472	5,279,744	309,290	2,009	1,783	38,521,858	1,294,472	10,241,591	1,607,481	7,189

Table B.14 – Tailored kernel binary statistics for Linux kernel v3.10 – v3.12

List of Acronyms

API	application programming interface
AST	abstract syntax tree
CI	continuous integration
CPP	C preprocessor
HID	human interface device
ID	identifier
KVM	kernel virtual machine
LKM	loadable kernel module
PC	program counter
RegExp	regular expression
SAT	(boolean) satisfiability problem
SmPL	semantic patch language

List of Figures

List of Listings

List of Tables

References

[1] *AspectC++*. Project Homepage. URL: http://www.aspectc.org/ (visited on 08/02/2014).

[2] *BasicLinux Homepage*. Website. URL: http://distro.ibiblio.org/baslinux/ (visited on 07/13/2014).

[3] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 41–41. URL: http://dl.acm.org/citation.cfm?id=1247360.1247401.

[4] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. "A Study of Variability Models and Languages in the Systems Software Domain." In: *IEEE Transactions on Software Engineering* 39.12 (2013), pp. 1611–1640. ISSN: 0098-5589. DOI: 10.1109/TSE.2013.34.

[5] Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez, Lluis Vilanova, Enric Morancho, and Nacho Navarro. *Building a Global System View for Optimization Purposes*. workshop. Boston, USA, June 2006. URL: http://personals.ac.upc.edu/rbertran/pdfs/wso-wiosca.pdf.

[6] Manfred Broy. "Challenges in Automotive Software Engineering." In: *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. (Shanghai, China). New York, NY, USA: ACM Press, 2006, pp. 33–42. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134292.

[7] *BusyBox Project Homepage*. URL: http://www.busybox.net/ (visited on 05/11/2012).

[8] *CADOS: Configurability Aware Development of Operating Systems*. Research Group Homepage. URL: https://www4.cs.fau.de/Research/CADOS/ (visited on 07/22/2014).

[9] 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)*. New York, NY, USA: ACM Press, 2005, pp. 95–104. ISBN: 1-59593-018-3. DOI: 10.1145/1065910.1065925.

[10] Noam Chomsky. "On certain formal properties of grammars." In: *Information and Control* 2.2 (1959), pp. 137 –167. ISSN: 0019-9958. DOI: http://dx.doi.org/10.1016/S0019-9958(59)90362-6. URL: http://www.sciencedirect.com/science/article/pii/S0019995859903626.

[11] Gabriele Cirulli. *2048*. GitHub Project. URL: http://gabrielecirulli.github.io/2048/ (visited on 04/23/2014).

[12] *clang - C Language Family Frontend for LLVM*. Project Homepage. URL: http://clang.llvm.org/ (visited on 07/16/2014).

[13] *Coder for Raspberry Pi*. GitHub Project. URL: http://googlecreativelab.github.io/coder/ (visited on 12/13/2013).

[14] *CyanogenMod - Android Community Operating System*. Project Homepage. URL: http://www.cyanogenmod.org/ (visited on 07/15/2014).

[15] *Description of the unified diff format*. the GNU diff manual. URL: http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html (visited on 07/13/2014).

[16] 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). Ed. by Eduardo Santana de Almeida, Christa Schwanninger, and David Benavides. New York, NY, USA: ACM Press, 2012, pp. 21–30. ISBN: 978-1-4503-1094-9. DOI: 10.1145/2362536.2362544.

[17] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Understanding Linux Feature Distribution." In: *Proceedings of the 2nd AOSD Workshop on Modularity in Systems Software (AOSD-MISS '12)*. (Potsdam, Germany, Mar. 27, 2012). Ed. by Christoph Borchert, Michael Haupt, and Daniel Lohmann. New York, NY, USA: ACM Press, 2012. ISBN: 978-1-4503-1217-2. DOI: 10.1145/2162024.2162030.

[18] *Expect*. Project Homepage. URL: http://expect.sourceforge.net/ (visited on 07/13/2014).

[19] *Fiasco Project Homepage*. URL: http://os.inf.tu-dresden.de/fiasco/ (visited on 05/11/2012).

[20] *function tracer guts*. the Linux kernel documentation. URL: https://www.kernel.org/doc/Documentation/trace/ftrace-design.txt (visited on 07/13/2014).

[21] Paul Gazzillo and Robert Grimm. "SuperC: Parsing All of C by Taming the Preprocessor." In: *SIGPLAN Not.* 47.6 (June 2012), pp. 323–334. ISSN: 0362-1340. DOI: 10.1145/2345156.2254103. URL: http://doi.acm.org/10.1145/2345156.2254103.

[22] *GCC, the GNU Compiler Collection*. Project Homepage. URL: https://gcc.gnu.org/ (visited on 07/16/2014).

[23] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. "A genetic algorithm for optimized feature selection with resource constraints in software product lines." In: *Journal of Systems and Software* 84.12 (2011), pp. 2208 –2221. ISSN: 0164-1212. DOI: http://dx.doi.org/10.1016/j.jss.2011.06.026. URL: http://www.sciencedirect.com/science/article/pii/S0164121211001518.

[24] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, 2011, 683 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853.

[25] *Jenkins CI*. Project Homepage. URL: http://jenkins-ci.org/ (visited on 07/13/2014).

[26] *Kconfig*. the Linux kernel documentation. URL: https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt (visited on 07/21/2014).

[27] Brian W. Kernighan. *The C Programming Language*. Ed. by Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[28] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. "Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring." In: *Proceedings of the 20th Network and Distributed Systems Security Symposium*. (San Diego, CA, USA, Feb. 24–27, 2013). The Internet Society, 2013. URL: http://www.internetsociety.org/sites/default/files/03_2_0.pdf.

[29] 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), pp. 1093–1108. ISSN: 1016-2364.

[30] *LG E960 / Google Nexus 4 Technical Specification*. URL: http://www.lg.com/us/support/products/documents/Nexus4_One_sheeter.pdf (visited on 07/15/2014).

[31] Jörg Liebig, Christian Kästner, and Sven Apel. "Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code." In: *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*. AOSD '11. Porto de Galinhas, Brazil: ACM, 2011, pp. 191–202. ISBN: 978-1-4503-0605-8. DOI: 10.1145/1960275.1960299. URL: http://doi.acm.org/10.1145/1960275.1960299.

[32] *Lineo uLinux - Embedded Linux*. Publisher Homepage. URL: http://www.lineo.co.jp/modules/products/ulinux.html (visited on 07/13/2014).

[33] *Linux kernel coding style*. Linux kernel documentation. URL: https://www.kernel.org/doc/Documentation/CodingStyle (visited on 07/13/2014).

[34] *Linux Tiny*. Embedded Linux Wiki. URL: http://elinux.org/Linux_Tiny (visited on 07/09/2014).

[35] *LLVM Linux*. Project Homepage. URL: http://llvm.linuxfoundation.org/index.php/Main_Page (visited on 07/13/2014).

[36] *Onion Pi*. Adafruit Learning System. URL: http://learn.adafruit.com/onion-pi/ (visited on 09/27/2013).

[37] *OOM Killer*. Linux Memory Management. URL: http://linux-mm.org/OOM_Killer (visited on 07/13/2014).

[38] *OPcache*. PHP Documentation. URL: http://php.net/manual/en/book.opcache.php (visited on 07/13/2014).

[39] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. "Understanding Collateral Evolution in Linux Device Drivers." In: *SIGOPS Oper. Syst. Rev.* 40.4 (Apr. 2006), pp. 59–71. ISSN: 0163-5980. DOI: 10.1145/1218063.1217942. URL: http://doi.acm.org/10.1145/1218063.1217942.

[40] *PHP Hypertext Preprocessor*. Official Website. URL: http://www.php.net/ (visited on 07/13/2014).

[41] *QEMU Emulator User Documentation*. URL: http://qemu.weilnetz.de/qemu-doc.html (visited on 07/13/2014).

[42] *QEMU Internals*. URL: http://qemu.weilnetz.de/qemu-tech.html (visited on 07/13/2014).

[43] *Raspberry Pi Model B Technical Specification*. URL: http://elinux.org/RPi_Hardware (visited on 07/15/2014).

[44] Dennis M. Ritchie. “The Development of the C Language.” In: *SIGPLAN Not.* 28.3 (Mar. 1993), pp. 201–208. ISSN: 0362-1340. DOI: 10.1145/155360.155580. URL: http://doi.acm.org/10.1145/155360.155580.

[45] Andreas Ruprecht, Bernhard Heinloth, and Daniel Lohmann. “Automatic Feature Selection in Large-Scale System-Software Product Line.” In: *13th International Conference on Generative Programming and Component Engineering (GPCE ’14)*. New York, NY, USA: ACM Press, 2014, pp. 39–48. DOI: 10.1145/2658761.2658767. URL: https://www4.cs.fau.de/Publications/2014/ruprecht_14_gpce.pdf.

[46] Horst Schirmeier and Olaf Spinczyk. “Tailoring Infrastructure Software Product Lines by Static Application Analysis.” In: *Proceedings of the 11th Software Product Line Conference (SPLC ’07)*. IEEE Computer Society Press, 2007, pp. 255–260. ISBN: 0-7695-2888-0. DOI: 10.1109/SPLINE.2007.33.

[47] N. Siegmund, S.S. Kolesnikov, C. Kastner, S. Apel, D. Batory, M. Rosenmuller, and G. Saake. “Predicting performance via automated feature-interaction detection.” In: *Proceedings of the 34nd International Conference on Software Engineering (ICSE ’12)*. (Zurich, Switzerland). Washington, DC, USA: IEEE Computer Society Press, June 2012, pp. 167–177. ISBN: 978-1-4673-1067-3. DOI: 10.1109/ICSE.2012.6227196.

[48] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. “SPL Conqueror: Toward optimization of non-functional properties in software product lines.” English. In: *Software Quality Journal* 20.3-4 (2012), pp. 487–517. ISSN: 0963-9314. DOI: 10.1007/s11219-011-9152-9. URL: http://dx.doi.org/10.1007/s11219-011-9152-9.

[49] 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). Ed. by Eelco Visser and Jaakko Järvi. New York, NY, USA: ACM Press, 2010, pp. 33–42. ISBN: 978-1-4503-0154-1. DOI: 10.1145/1868294.1868300.

[50] Samaneh Soltani, Mohsen Asadi, Dragan Gašević, Marek Hatala, and Ebrahim Bagheri. “Automated Planning for Feature Model Configuration Based on Functional and Non-functional Requirements.” In: *Proceedings of the 16th International Software Product Line Conference - Volume 1*. SPLC ’12. Salvador, Brazil: ACM, 2012, pp. 56–65. ISBN: 978-1-4503-1094-9. DOI: 10.1145/2362536.2362548. URL: http://doi.acm.org/10.1145/2362536.2362548.

[51] Diomidis Spinellis. “Global Analysis and Transformations in Preprocessed Languages.” In: *IEEE Transactions on Software Engineering* 29.11 (Nov. 2003), pp. 1019–1030. ISSN: 0098-5589. DOI: 10.1109/TSE.2003.1245303. URL: http://www.spinellis.gr/pubs/jrnl/2003-TSE-Refactor/html/Spi03r.html.

[52] *systemd*. Project Homepage. URL: http://freedesktop.org/wiki/Software/systemd/ (visited on 07/13/2014).

[53] *SysVinit*. Project Homepage. URL: http://freshmeat.net/projects/sysvinit/ (visited on 07/13/2014).

[54] *Tailor HowTo*. VAMOS Undertaker Project. URL: http://vamos.informatik.uni-erlangen.de/trac/undertaker/wiki/TailorHowto (visited on 07/13/2014).

[55] Reinhard Tartler, Anil Kurmus, Bernard Heinloth, Valentin Rothberg, Andreas Ruprecht, Daniela Doreanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability.” In: *Proceedings of the 8th International Workshop on Hot Topics in System Dependability (HotDep ’12)*. (Los Angeles, CA, USA). Berkeley, CA, USA: USENIX Association, 2012, pp. 1–6.

[56] 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). Ed. by Christoph M. Kirsch and Gernot Heiser. New York, NY, USA: ACM Press, Apr. 2011, pp. 47–60. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966451.

[57] *Tcl Developer Site*. Official Website. URL: http://www.tcl.tk/ (visited on 07/13/2014).

[58] *The Apache HTTP Server Project*. Official Website. URL: http://httpd.apache.org/ (visited on 07/13/2014).

[59] *The Core Project: Tiny Core Linux*. Project Homepage. URL: http://tinycorelinux.net (visited on 07/13/2014).

[60] *The LLVM Compiler Infrastructure Project*. Official Website. URL: http://llvm.org/ (visited on 07/16/2014).

[61] *Tor (previously known as The Onion Router) Project*. Official Website. URL: https://www.torproject.org/ (visited on 01/22/2014).

[62] *Upstart event-based init daemon*. Project Homepage. URL: http://upstart.ubuntu.com/ (visited on 07/13/2014).

[63] *VAMOS: Variability Management in Operating Systems*. Research Group Homepage. URL: https://www4.cs.fau.de/Research/VAMOS/ (visited on 07/22/2014).