On Synthesising Linux Kernel Module Components from Coq Formalisations

Mario Frank

A Dissertation Presented for the Degree of
Doctor Rerum Naturalium
in Theoretical Computer Science

University of Potsdam
Faculty of Science
Institute of Computer Science

Potsdam, Germany
2024-05-31

Doctoral Committee:

Main Supervisor, 1. Reviewer:	Prof. Dr. Christoph Kreitz
Second Supervisor, 3. Reviewer:	Prof. Dr. Anna-Lena Lamprecht
External Reviewer:	Prof. Andrew W. Appel, PhD. (Princeton University)
Additional Members:	Prof. Dr. Bettina Schnor
	Prof. Dr-Ing. Ulrike Lucke
	PD Dr. habil. Henning Bordihn

Frank, Mario
`mario.frank@uni-potsdam.de`
On Synthesising Linux Kernel Module Components from Coq Formalisations
Dissertation, Institute of Computer Science, Faculty of Science
University of Potsdam, 2024

Published online on the
Publication Server of the University of Potsdam:
`https://doi.org/10.25932/publishup-64255`
`https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-642558`

Abstract

This thesis presents an attempt to use source code synthesised from Coq formalisations of device drivers for existing (micro)kernel operating systems, with a particular focus on the Linux Kernel.

In the first part, the technical background and related work are described. The focus is here on the possible approaches to synthesising certified software with Coq, namely the extraction to functional languages using the Coq extraction plugin and the extraction to Clight code using the CertiCoq plugin. It is noted that the implementation of CertiCoq is verified, whereas this is not the case for the Coq extraction plugin. Consequently, there is a correctness guarantee for the generated Clight code which does not hold for the code being generated by the Coq extraction plugin. Furthermore, the differences between user space and kernel space software are discussed in relation to Linux device drivers. It is elaborated that it is not possible to generate working Linux kernel module components using the Coq extraction plugin without significant modifications. In contrast, it is possible to produce working user space drivers both with the Coq extraction plugin and CertiCoq. The subsequent parts describe the main contributions of the thesis.

In the second part, it is demonstrated how to extend the Coq extraction plugin to synthesise foreign function calls between the functional language OCaml and the imperative language C. This approach has the potential to improve the type-safety of user space drivers. Furthermore, it is shown that the code being synthesised by CertiCoq cannot be used in kernel space without modifications to the necessary runtime. Consequently, the necessary modifications to the runtimes of CertiCoq and VeriFFI are introduced, resulting in the runtimes becoming compatible components of a Linux kernel module. Furthermore, justifications for the transformations are provided and possible further extensions to both plugins and solutions to failing garbage collection calls in kernel space are discussed.

The third part presents a proof of concept device driver for the Linux Kernel. To achieve this, the event handler of the original PC Speaker driver is partially formalised in Coq. Furthermore, some relevant formal properties of the formalised functionality are discussed. Subsequently, a kernel module is defined, utilising the modified variants of CertiCoq and VeriFFI to compile a working device driver. It is furthermore shown that it is possible to compile the synthesised code with CompCert, thereby extending the guarantee of correctness to the assembly layer. This is followed by a performance evaluation that compares a naive formalisation of the PC speaker functionality with the original PC Speaker driver pointing out the weaknesses in the formalisation and possible improvements. The part closes with a summary of the results, their implications and open questions being raised.

The last part lists all used sources, separated into scientific literature, documentations or reference manuals and artifacts, i.e. source code.

Zusammenfassung

Die vorliegende Dissertation präsentiert einen Ansatz zur Nutzung von Quellcode, der aus der Coq-Formalisierung eines Gerätetreibers generiert wurde, für bestehende (Mikrokernel-)Betriebssysteme, im Speziellen den Linux-Kernel.

Im ersten Teil erfolgt eine Beschreibung der relevanten technischen Aspekte sowie des aktuellen Forschungsstandes. Dabei liegt der Fokus auf der Synthese von funktionalem Code durch das Coq Extraction Plugin und von Clight Code durch das CertiCoq Plugin. Des Weiteren wird dargelegt, dass die Implementierung von CertiCoq im Gegensatz zu der des Coq Extraction Plugin verifiziert ist, wodurch sich eine Korrektheitsgarantie für den generierten Clight Code ableiten lässt. Darüber hinaus werden die Unterschiede zwischen User Space und Kernel Space Software in Bezug auf Linux-Treiber erörtert. Unter Berücksichtigung der technischen Einschränkungen wird dargelegt, dass der durch das Coq Extraction Plugin generierte Code ohne gravierende Anpassungen der Laufzeitumgebung nicht als Teil eines Kernel Space Treibers nutzbar ist. Die nachfolgenden Teile der Dissertation behandeln den Beitrag dieser Arbeit.

Im zweiten Teil wird dargelegt, wie das Coq Extraction Plugin derart erweitert werden kann, dass typsichere Aufrufe zwischen den Sprachen OCaml und C generiert werden können. Dies verhindert spezifische Kompilationsfehler aufgrund von Typfehlern. Des Weiteren wird aufgezeigt, dass der durch CertiCoq generierte Code ebenfalls nicht im Kernel Space genutzt werden kann, da die Laufzeitumgebung technische Einschränkungen verletzt. Daher werden die notwendigen Anpassungen an der vergleichsweise kleinen Laufzeitumgebung sowie an VeriFFI vorgestellt und deren Korrektheit begründet. Anschließend werden mögliche Erweiterungen beider Plugins sowie die Möglichkeit der Behandlung von fehlschlagenden Aufrufen der Garbage Collection von CertiCoq im Kernel Space erörtert.

Im dritten Teil wird als Machbarkeitsstudie im ersten Schritt der Event-Handler des Linux PC Speaker Treibers beschrieben und eine naive Coq-Formalisierung sowie wichtige formale Eigenschaften dargelegt. Dann wird beschrieben, wie ein Kernel-Modul und dessen Kompilation definiert werden muss, um einen lauffähigen Linux Kernel Treiber zu erhalten. Des Weiteren wird erläutert, wie die generierten Teile dieses Treibers mit dem verifizierten Kompiler CompCert übersetzt werden können, wodurch auch eine Korrektheit für den resultierenden Assembler-Code gilt. Im Anschluss erfolgt eine Evaluierung der Performance des aus der naiven Coq-Formalisierung generierten Codes im Vergleich zum originalen PC-Speaker Treiber. Dabei werden die Schwächen der Formalisierung sowie mögliche Verbesserungen diskutiert. Der Teil wird mit einer Zusammenfassung der Ergebnisse sowie der daraus resultierenden offenen Fragen abgeschlossen.

Der letzte Teil gibt eine Übersicht über genutzte Quellen und Hilfsmittel, unterteilt in wissenschaftliche Literatur, Dokumentationen sowie Software-Artefakte.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

In recent decades, numerous initiatives have been undertaken with the objective of enhancing the security and reliability of software. These efforts are justified by the fact that errors in software are not only annoying but also can be dangerous in terms of monetary loss and, in the most severe cases, even loss of human life. Approaches as unit testing [25] have shown to be highly applicable in software engineering and do have a positive impact on software quality. However, it is also true that these approaches are limited since only the potential errors which the engineer is aware of will be covered.

In this context, software verification can be applied to certify the properties of software, based on the source code. With appropriate tool support, it is even possible to formalise the functionality of a software and then generate source code that is guaranteed to be conforming to the specification. While both approaches are nowadays commonly used for ordinary programs, the most critical portions of software are those running in privileged mode directly on hardware, i.e. the operating system. Attempts have been made to enhance the safety of operating systems like Linux by introducing memory-safe programming languages like Rust [30]. While this is a step in the right direction, it does not guarantee correctness. Moreover, there is much research been and being conducted on the verification of operating systems and device drivers. Nevertheless, numerous questions remain unanswered.

In this thesis, one of those questions will be raised and answered: namely, whether it is possible to synthesise[1] fully functional Linux Kernel components from functionality specifications using a proof assistant, here Coq [10]. The rationale for choosing Coq is mostly originated from the VerSeCloud [21] project which shares common ground with the aims of this thesis.

[1]The term "synthesis" is used here to distinguish it from normal compilation, and to provide a more general term for compilation and extraction. The main motivation stems from the fact that proof assistants like Coq allow users to define software in a specification language that has a higher level of formality and clearer semantics than programming languages like C or OCaml.

The main contributions of this thesis are to show that the

- code generated by the Coq extraction plugin [38] cannot be usilised for Linux kernel space drivers without significant modifications to the runtime being necessary while it can be used for user space device drivers
- Coq extraction plugin can be extended to generate foreign function calls, thereby enhancing the type-safety of those calls
- code generated by the CertiCoq [2] plugin can be utilised for kernel space drivers provided necessary modifications to the relatively small runtime
- modifications to the CertiCoq runtime can be proven to be correct
- same correctness guarantees apply to the code generated by CertiCoq for both user and kernel space
- code generated by CertiCoq can be compiled by the certified compiler CompCert for kernel space which implies that the correctness guarantees can be extended to the assembly level for kernel space as well.

The thesis commences with an introduction to the pertinent aspects of and research on software verification, software synthesis and the difference between ordinary (i.e. user space) programs and Linux Kernel (kernel space) components. Additionally, general considerations on combining generated with manually written code are given, and the part closes with a discussion of the relevant aspects. In the second part, the suitability of the different synthesis pipelines of Coq, namely the Coq extraction plugin and CertiCoq, is discussed. Simultaneously, the necessary steps and transformations to make the generated source code compatible components of a Linux Kernel module are presented. This part concludes with a discussion of the results, poses new open questions and leads to the third part. In the third part, a proof of concept will be presented which demonstrates that it is indeed possible to generate fully functioning Linux Kernel components given adapted runtimes for that purpose. Furthermore, it will be shown that it is possible to apply certified compilation to the generated kernel space components. Thus, it will be confirmed that the correctness guarantees given by the used synthesis and compilation do hold for both user and kernel space software being generated by Coq. Furthermore, it will be argued that the same results that hold for the Linux Kernel can be applied to other comparable operating systems. The part concludes with a summary of the results, their implications and open questions that could not be addressed in the course of this thesis. The last part lists all the literature, documentation, specifications and source code that were used.

Part I

Technical Background and Related Work

Chapter 2

Device Drivers

This chapter examines the distinctions between user space and kernel space software in general and device drivers in particular. These differences are of vital importance when formalising and synthesising functionality, as different Application Programming Interfaces (APIs) have to be used depending on the target. In the following section, the main differences between user space and kernel space software are described. Then, specialities on user space and kernel space device drivers are introduced. It is important to note that the original de facto reference manual [16] on Linux Device Drivers, has not been updated for almost two decades despite significant developments in the Linux Kernel and, in particular, the structure of device drivers. Consequently, for the most up-to-date information on these topics, the documentation of the Linux Kernel itself is consulted and cited.

2.1 User Space vs. Kernel Space Software

The distinction between user space and kernel space software is multifaceted, even when both are implemented in C. The primary distinction lies in the capability of user space software to utilise the C standard library, commonly referred to as the libc. There are various implementations of the libc, including the glibc [87] and the uClibc-ng [102]. All of these libraries provide fundamental functionality for user space software, including the ability to print to a stream (e.g. the console) by `fprintf`, to allocate and deallocate heap memory by `malloc` and `free`, or to instantiate threads using `pthreads`. These functionalities internally trigger a system call to the Linux Kernel, i.e. the named functions are interfaces to Linux Kernel functionality. In contrast, kernel space software, such as kernel modules, does not run in user space and normally does not have access to the C standard library. Consequently, they must communicate with the Linux Kernel directly via the more low-level kernel standard library.

For example, the Linux Kernel does not include a `printf` or `fprintf` function. Instead, for example the `printk` [71] function from the header `linux/printk.h` can be employed to print messages into the kernel log files. The `printk` function requires a log level as input, typically `KERN_INFO`, a format string and, if the format string includes placeholders, additionally a colon-separated list of arguments. The primary distinctions between `printf` and `printk` are the log level and the different header files. In addition to `KERN_INFO`, numerous other log levels can be used, depending on the severity of the message. Furthermore, there is a special log level for continuing the preceding message (`KERN_CONT`). However, this log level is discouraged in most situations. It should only be used in the early boot process by core or architecture-specific code as it may not be SMP[1] safe otherwise [110, Line 20–22]. Furthermore, `printk` is subject to inherent limitations, for example, floating-point conversion specifiers are not supported. In general, floating-point operations in kernel space can be problematic. It is therefore recommended to avoid floating-point operations since the Floating Point Unit context must be saved and restored manually and context switches must be avoided [66, 104]. An alternative would be to use fixed point arithmetic instead. Another limitation is the absence of an alternative for `fprintf`, i.e. for printing into files. While it is theoretically possible to achieve this [33], it is strongly discouraged since, for example, the directory layout is not necessarily fixed and consistent between Linux distributions.

Furthermore, memory allocations [70] are also quite different in kernel space. One aspect is that when a kernel module allocates memory, it has to be stated whether this memory is used only by the module itself or potentially also by some user space software. This is determined by whether the module allocates the memory for itself (private) or shared. Consequently, kernel modules employ the functions `kmalloc`, `kmalloc_array` and `kzalloc` from the Linux `linux/slab.h` file instead of using malloc from the libc. All those functions have in common that they have, apart from the memory size that shall be allocated, an additional parameter `GFP_FLAGS` (Get Free Page Flags). That parameter states how the memory shall be allocated. For example, `GFP_USER` signifies that the memory to be allocated shall not be movable and that the kernel must have direct access to it. Furthermore, `kzalloc` initialises the memory with zeros to ensure that no previously stored data is leaked. Finally, `kmalloc_array` is specifically designed to facilitate the allocation of arrays in a more secure manner than multiplying the element size with the cell count - a functionality that does not exist in the user space libc. The release of memory is also carried out by `kfree` instead of `free`. However, there are variant functions for freeing memory that must be used depending on how the memory was allocated.

[1]Symmetric multiprocessing, i.e. multiple (identical) processors using shared memory

While dynamic allocations, i.e. heap allocations, can be employed in principle without problems, there is a rather strict limit on the stack size. According to the "basic kernel hacking rules" [66], the stack is limited to 3K to 6K for most 32-bit architectures and about 14K for most 64-bit architectures. In the case of the the x86_64 architecture, for instance, the stack size for an active thread is limited to four pages where a page has a size of 4K [67] resulting in a maximum stack size for each active thread of 16K. While the user space stack limits can be set by `ulimit` to a verbatim value [77], there is no such functionality for kernel space since the stack size is defined during the compilation of the kernel. Furthermore, exceeding the stack size in user space produces a segmentation violation, which results in the termination of the program. In kernel space, exceeding the stack space is much more serious, as kernel space software is part of the kernel and the violation will not be handled. This can lead to overwriting vital parts of the system memory, resulting in a system freeze. Thus, deep recursion has to be avoided.

It is also important to note that user space code can use the `exit` [93, §7.22.4.4] call from the `stdlib.h`, e.g. when encountering an error. This does not hold for kernel modules since the `exit` function is a libc functionality. Kernel modules are expected to catch encountered errors and return an error value from the function implementing the event handling, for example. While there is some kernel space functionality to exit from a process comparable to the user space `exit`, namely the function `do_exit` [114], this function can only be used for kernel threads (`kthread`). Thus, it would be in principle possible to wrap the call to synthesised code into a `kthread` and call `do_exit` on error. However, kernel threads are meant to be used as workers, e.g. for handling network packages in parallel. Consequently, this would be a misuse and would additionally result in the production of unnecessary computational overhead due to the scheduling of threads. Comparably to `exit`, the "non-local jumps" (`setjmp` and `longjmp`) [93, §7.13.1.1 & §7.13.2.1] are defined in the user space standard library, e.g. glibc [86]. Those functions can be used to set a jump target position in the code and jump back, respectively. Although the functions are defined in assembly code [111], they are not used in kernel space code[2] [108], indicating that they are not usable.

A commonly used functionality in C code are assertions [85]. These check whether an expected condition holds and if it does not, an error is emitted with details on the failed assertion including the exact position in the source code. Additionally, the process (i.e. the program) is stopped. However, assertions are automatically eliminated when compiling C code with the definition `NDEBUG`. For kernel space software, a comparable functionality does exist with the macro `BUG_ON` [113, Line 71]. However, the main problem here is that the process that would be stopped when this assertion fails is the Linux Kernel itself. Consequently, an error

[2]Apart from uses in the `tools` subdirectory where user space functionality is located.

condition would not only stop the kernel module but the complete system (with a "Kernel Panic") which is discouraged [105]. While assertions are automatically removed in user space when using `NDEBUG` as a compilation definition, this is not possible in kernel space. Instead, the use of `BUG_ON` produces a Kernel Panic if the definition `CONFIG_BUG` is set and leads to an infinite loop, otherwise [113, Line 159]. Both effects are undesirable when considering safety-critical systems, such as those used in the operation of trains. A failing control driver in a train has the potential to be a risk to life. Consequently, errors in kernel space should neither result in a panic nor in an infinite loop. Instead, errors should be propagated and returned by an error value by the most outer function of the module.

All of the aforementioned limitations[3] apply for all parts of the Linux kernel, including device drivers. Furthermore, Linux Kernel components as drivers are normally no standalone programs. Normal programs are started with a `main` function (in C), potentially together with some command line arguments. They frequently perform a single operation[4], for example converting one image format into another. In contrast, device drivers rather register themselves (e.g. to the kernel) to serve as event handlers [16, Ch. 2, "Kernel Modules Versus Applications"], for example. Thus, they have a more concrete life cycle, i.e. they can be started and stopped. In the following section, the distinction between (Linux) kernel space and user space drivers will be described.

2.2 Kernel Space Drivers

Linux device drivers can be constructed in two distinct ways: as loadable modules or as part of the monolithic kernel. In the context of this thesis, the former variant is considered since the proof of concept is demonstrated on a loadable module. Linux kernel space drivers follow a driver model which is defined as a static structure (as shown in Listing 2.1) and every driver must at least define its name and the bus fields it uses [62]. Nevertheless, according to [62], not all existing drivers adhere to this model, and it will not be possible to convert all drivers fully to this driver model.

In addition to the necessary definitions, callbacks can be defined. These are automatically called on specific events. For example, the callbacks `.suspend` and `.resume` are called when the system enters a low power state (e.g. standby) and when it resumes from this state. The `.probe` callback is called when the driver is started (via `insmod` or `modprobe`), and then binds the driver to the device. The `.remove` callback is called when the driver is unloaded (via `rmmod`) to unbind the driver and apply any necessary cleanup routines.

[3]Other limitations may also apply but are not relevant for the proof of concept.

[4]which may be quite complex, nevertheless.

```
static struct device_driver some_driver = {
  .name    = "driver_name",
  .bus     = &pci_bus_type,

  .probe   = my_probe,
  .remove  = my_remove,
  .suspend = my_suspend,
  .resume  = my_resume,
};
```

Listing 2.1: Device driver definition, adapted example from [62].

However, the Linux Kernel supports different classes of drivers that may support additional callbacks. For example, the platform device driver model [72] supports additional callbacks for `shutdown`, `suspend_late` and `resume_early`. The `shutdown` callback is triggered when the system is to be shut down, and the others are used to support extended power management phases [63]. That is, `resume_early` is called before `resume`, for example to perform preparatory configurations on the device [63, "Leaving System Suspend"] or to undo configurations applied by `suspend_late` [63, "Entering System Suspend"]. Depending on the device type, additional functionality may be used. The Linux PC Speaker driver [116], for example, is a platform device driver and manages an input device [65]. This provides the ability to bind to the physical device `isa0061/input0` during the probing phase, and to set the `event` callback to a function that handles incoming events.

Once a device driver has been implemented for the Linux Kernel, it can be built as an external module [60]. This can be done using the `kbuild` build system and requires the definition of a `Makefile` [82]. To compile a module, the `Makefile` must at least define which module to build by specifying the name of the resulting object file in a variable declaration, e.g. `obj-m := my_driver.o`. The suffix of the variable defines whether it is a built-in driver (`y`) or a loadable module (`m`). In this case, the file `my_driver.c` (or `my_driver.S`) is automatically compiled into `my_driver.o`, and the compilation results in a kernel object named `my_driver.ko`. However, it is possible to add other necessary objects to the `my_driver` module by specifying the variable `my_driver-y := f1.o`. Here the prefix `my_driver` is the relevant aspect, i.e. the variable must have this prefix, while the suffix `y` can also be set to `objs` for composite host programs [60, Host Program Support]. This allows the toolchain to recognise that any named object file must be compiled from an existing C file of the same name, but with a `.c` suffix. To speed up compilation of the Linux Kernel or module, `.cmd` files have been introduced [22]. These files are created by the compilation toolchain for each `f.c` that is compiled into `f.o`, and contain all the compilation parameters and additional information. This allows the toolchain to detect if a file has already been built, and also to detect changes in the compilation parameters and rebuild in that case [61, "Command change detection"].

One notable option, which will become relevant in Section 9, is the ability to use binary blobs [60, §3.3], i.e. shipped object files. This option can be used to link precompiled object files, e.g. files built with another compiler or for proprietary components for which no source code is available. For example, if the module requires a file called `prims.o`, it is possible to use `ocamlopt` to compile the file `prims.ml` as `prims.o_shipped`. Here the file is indeed a normal object file. But the suffix `_shipped` is recognised by the toolchain and it will copy the file `prims.o_shipped` to `prims.o`, thus resolving the dependency. But for shipped objects, the compilation toolchain has no information about how they were compiled, because it did not compile them itself. Furthermore, it expects that for each `f.o` file, there is a `f.o.cmd` file after compilation, which leads to an error when using a shipped object. To circumvent this, the `.cmd` file must be created separately for the shipped objects, but it can be empty[5].

The compilation of a kernel module involves many compilation parameters which greatly affect the compilation itself. A discussion of all possible parameters is beyond the scope of this thesis, as the parameters depend on many aspects such as the target architecture and the configuration of the kernel itself, as can be seen in the Makefile [118]. Therefore, the relevant compilation parameters will be discussed later in the respective parts of this thesis.

2.3 User Space Drivers

While Linux device drivers are kernel space software, it is also possible to implement driver functionality in user space. In this case, the drivers are normal programs that register as services. This can be a good solution when a device is used exclusively by one process [73, 94]. However, while user space drivers do exist for Linux [137] [1, 18], most drivers are implemented in kernel space. For operating systems such as the L4Re Operating Systems Framework [133] which consists of a small microkernel [134] and a user space runtime environment, drivers are implemented almost entirely in user space. Even complex drivers such as the NVME [135] block device driver are implemented entirely in user space, which is unusual for Linux. Similar to Linux, all drivers register themselves to the kernel as servers and respond to potentially multiple clients communicating with the NVME server. The advantage of user space drivers is that the limitations of kernel space software, especially stack size, do not apply.

[5] This approach and the origin of the error message have been discussed for example in [98].

Chapter 3

Verification and Synthesis of Software

3.1 Verification of Software

There have been many projects attempting to prove the correctness of software, and there are many possible approaches. As mentioned previously, unit testing is not a proof of correctness. Sophisticated techniques such as static analysis [12] are able to find specific errors in existing source code, even in the absence of input data. Model checking [8] goes further by reasoning about the state of a program. Proof assistants such as Coq [10] and Isabelle [45] allow both the specification of software and the proof of its properties. However, they differ in their theoretical foundations and specification languages.

Using separation logic [51], which is an extension of Hoare logic [26], it is possible to reason about mutable data structures or (the ownership of) pointers. Since operating systems like the Linux Kernel contain many functions that manipulate memory, formalisms like separation logic can be useful for proving their correctness. Research programmes such as DeepSpec [6] play an important role here, as they attempt to verify a wide variety of aspects of software in general and operating systems in particular, and in many of the projects that are part of DeepSpec, formalisations and verifications have been carried out in the Coq Proof Assistant.

For example, the Verified Software Toolchain (VST) [5] uses higher-order separation logic to reason about software properties while using proofs in Coq. This toolchain has been used for several related applications, such as reasoning about the communication [40] of the verified operating system CertiKOS [23], which is also part of the DeepSpec programme. Also, the verified foreign function interface VeriFFI [32] [144] between Coq and C uses the VST. Other parts of DeepSpec include the verification of embedded systems based on the RISC-V [106] architecture, including their software, i.e. compilers, drivers and applications [19]. The most relevant DeepSpec project for this thesis is CompCert [35, 36, 37, 29]. CompCert is a compiler for the C language and has been verified to correctly compile the preprocessed C code up to the generation of assembly code. As an intermediate step, the compiler transforms the C code into the Clight [11] fragment of C, which has a clearer mechanised semantics. CertiCoq [2] closes the gap between a Coq formalisation and the Clight code, i.e. it is a compiler that generates Clight code that is equivalent to the Coq formalisation in terms of the implemented functionality. CertiCoq is described in more detail in the next section.

In fact, CertiKOS is not the only operating system that has been formally verified. For example, the seL4 [31] microkernel operating system has been verified in Isabelle/HOL, based on its C source code. There is also research research into the verification of the seL4 Core Platform [49] and device drivers based on seL4. For example, a BSD device driver has been automatically translated to Isabelle and then verified in Isabelle/HOL [44].

3.2 Synthesis of Certified Software

When a formalisation of a piece of software is available, it is desirable to synthesise source code that is correct, i.e. that conforms to a formal specification. The formalisation in Coq usually consists of a functional model, i.e. a functional representation of the algorithm in the Gallina language [91] [10], and a formal specification, which defines the high-level properties that the functional model should have. However, it is also possible to postulate the existence of a function that satisfies the high-level properties. To prove its existence, such a function must be derived or given. This function, also called a witness, can then be extracted by projection and stored as a Coq function. In this way, the functional model can be derived from the formal specification. In any case, synthesis is done using the functional model. This section describes how to synthesise the correct code from a formalisation. Extraction into functional languages using the Coq extraction plugin is described in Subsection 3.2.1, while the Subsection 3.2.2 describes the generation of Clight [11] code by the CertiCoq plugin and gives a comparison with the Coq extraction plugin in terms of trustworthiness and performance. The considerations on the synthesis of certified software are completed by a description and comparison of other proof assistants' synthesis approaches in Subsection 3.2.3 and general considerations about combining generated code with hand-written code in Subsection 3.2.4.

3.2.1 The Coq Extraction Plugin

The Coq extraction plugin [38] supports the synthesis of functional code from Coq formalisations, and the languages currently supported OCaml, Haskell and Scheme[1]. Compared to the previous version [50] of the Coq extraction plugin, which only supported a subset of the Coq specification language, the new extraction mechanism introduced in Coq 7.3 has closed this gap. But even more importantly, the new extraction was introduced together with a proof of a theoretical model of the new extraction.

However, while the theoretical model may be correct, this does not mean that the plugin itself is implemented correctly. Therefore, the plugin itself cannot be considered verified and fully trustworthy. Nevertheless, the plugin is relevant to the scope of this thesis and will therefore be described in more detail. The synthesis of functional code from Coq formalisations can be done, for example, by using the Coq command **`Extraction`** `@qualid` or **`Recursive Extraction`** `@qualid` [92, "Generating ML Code"]. This will extract the term specified by `@qualid` in the selected functional language, by default OCaml. Normally these commands only display the extracted terms in Coq. For the former variant, however, a version is provided that allows specifying an output file name, which leads to the synthesis of the terms in a pair of `ml` and `mli` files.

It is also possible to define ML code that implements functions for axiomatic Coq functions using the Coq command **`Extract Constant`** `@qualid` [92, "Realizing axioms"], while specifying a mapping from `@qualid` to the ML function. Similarly, it is possible to extract inductive types or constructors to specific ML types or constructors [92, "Realizing inductive types"].

However, it is not possible to call foreign (C) functions from the synthesised code without first extracting an ML function, which must then explicitly call C code via the foreign function interface. And these calls are particularly relevant to the goal of synthesising device driver functionality since this usually involves calling C functions. Nevertheless, the Coq extraction plugin can be a viable option for synthesising device driver components with at least some limited guarantees of correctness, although not being fully verified.

Once the source code has been synthesised using the Coq extraction plugin, the code still needs to be compiled to produce an executable program. Both Haskell and OCaml include rather large runtime libraries that need to be linked with the synthesised code [90, 100] when producing binary files. But this also means that the respective runtimes have to comply with kernel space restrictions. And there are many problematic places in the runtime, taking OCaml as an example[2].

[1]In the referenced article, only OCaml and Haskell were supported extraction targets, and Scheme extraction was a work in progress.

[2]And similar examples can easily be found for Haskell, e.g. in the file `rts/Weak.c` [101].

For example, the OCaml runtime uses the `exit` function wrapped in the `caml_do_exit` [124] function. In addition, assertions are emulated by the OCaml `caml_failed_assert` [119] wrapper function, which uses the `abort` function from libc. OCaml also has its own memory management [121], which uses the libc `malloc` function to allocate heap memory. Finally, the runtime includes floating-point operations [120, 122], including the printing of floating-point expressions.

These examples already show that making the runtime fully compatible with the kernel space would require huge changes. And while it is possible to achieve this, the benefits would be doubtful. Still, the trustworthiness would be broken by the compiler, as neither `ocamlc`, nor `ocamlopt`, nor `gch` are verified. Thus, using OCaml or Haskell as extraction languages for Linux Kernel modules is not a viable approach, while it is for user space drivers.

The Scheme programming language can be compiled using a number of different compilers [76], and there are several dialects of the language. Also, there are compilers that produce C code [139] [74, 78, 81], JVM code [139] [75] and native code [145] [88] directly. Some of these require a runtime to compile the native code [138, 140] [74, 79, 80, 81], where the same issues arise as with OCaml. For example, the code generated by the Chicken [74] compiler needs the `runtime.c`, which also contains the libc functionality used. And this applies to some extent to all named compilers targeting C. This is not really surprising, since basic functionality such as memory management and IO is usually done with libc functionality[3]. Apart from the runtimes, none of these named implementations are verified. Therefore, the correctness of the compilation is not guaranteed. While a verified compiler did exist for VLISP [24], up-to-date information on its status is not easily findable, making it difficult to assess its usability.

Thus, the target languages of the Coq extraction plugin do not appear to be viable options for kernel space use, but still viable options for user space drivers.

3.2.2 The CertiCoq Plugin

As mentioned earlier, the correctness of the Coq extraction plugin has not been proven in terms of the implementation. Here the CertiCoq [2, 9] plugin promises a much stronger correctness guarantee, i.e. the complete original pipeline from the Gallina specification language to the generated Clight code has been verified. The verification and parts of the pipeline are based on MetaCoq [53] and Template-Coq [39], while the elimination of Coq-specific sorts (i.e. `Type` and `Prop`) is based on the metatheory of the Calculus of Inductive Constructions [15]. Furthermore, the `Type` and `Prop` erasure was proved to be correct [54]. The continuation pass-

[3]Loko Scheme [145] can produce freestanding binaries, but they then contain an assembly code runtime.

ing style (CPS [55]) transformation is based on [4], and optimisations on the CPS representation were also shown to be correct [47, 7]. All these results provide a strong argument for the trustworthiness of CertiCoq's generation of Clight code from a Coq specification.

Another advantage of this synthesis is that the resulting code can be compiled with GCC [84] and clang [103] but even more importantly with CompCert, which itself has been verified to be correct as described in Section 3.1. Thus, it is possible to obtain a full correctness guarantee for the code generated by the CertiCoq pipeline and extend it to assembly code generation when using CompCert. Although the preprocessing of CompCert is not verified, this is not a major problem for the code generated by CertiCoq, as the use of preprocessing commands to generate the code is minimal.

Regardless of which compiler is used to compile the generated Clight code, additional source files are required [99]. These are some headers (`values.h`, `m.h`, `config.h`) that are required because the CertiCoq code generation uses the OCaml objects formats [41, Ch. 20], including the definition of the `value` type. In addition, and most importantly, garbage collection is required because Clight is a subset of C and therefore memory allocations must be managed manually. The garbage collection used is a generational garbage collector that has been verified using the CertiGraph [58, 52] framework. The corresponding GitHub repository [125] contains both the original source code of the garbage collector and the formalisation and verification of its functionality. Together, the named headers and the garbage collection form the runtime for any Clight program. And this is another advantage of CertiCoq, i.e. the runtime is small compared to those of OCaml and Haskell.

However, there are two drawbacks. First of all, it has been shown in [9] that the runtime performance of source code synthesised by CertiCoq and compiled with CompCert is significantly weaker compared to synthesised OCaml code when compiled natively (with `ocamlopt`). More specifically, for the selected examples, the runtime performance is typically between that of comparable interpreted OCaml and natively compiled OCaml code. However, this drawback can be reduced with improvements in the code generation or with additional optimisation steps, such as those introduced in [46, 48], for example. For the scope of this thesis, the second drawback is much more relevant. When the Coq formalisation `prog.v` is transformed (with `CertiCoq Compile`), the files `prog.c` and `prog.h` are synthesised. These files do not directly use any libc functionality. However, there are header files for primitive operations that are imported by default when using the import clause `From CertiCoq.Plugin Require Import CertiCoq`. However, the includes can be omitted by importing the `Loader` module instead[4]. This way the synthesised code can be used as a component of a kernel module without modifications to the

[4]This hint was given in private communication by Matthieu Sozeau from the CertiCoq team.

internal functionality. This more or less also holds for the glue code that provides transformations between C and Coq types and can be generated by the command `CertiCoq Generate Glue`. Only minor modifications to this code are necessary to make it usable in kernel space, as will be shown in Subsection 6.2.2. But the garbage collection utilises much functionality that is not usable in kernel space. This includes the libc functionality discussed in Section 2.1 but also the loss of control flow by `exit`. Nevertheless, these aspects can be resolved, and will be in Section 6.1.

3.2.3 Related Synthesis Approaches

The approach most closely related to this thesis is the verified extraction of Coq terms to OCaml, published as a preprint in 12/2023 [20]. This work attempts to address the most critical aspect of the current Coq extraction plugin, namely correctness. The new pipeline defines the extraction in Coq itself rather than in OCaml, using MetaCoq and extracting to an intermediate language of the OCaml compiler presented in [17], since OCaml has no formal specification. While the verified variant is a significant improvement for the synthesis of user space driver components, the lack of compatibility of the OCaml runtime with kernel space drivers remains. Also, the new extraction does not include the synthesis of foreign function calls to C.

In [3], the synthesis of Gallina terms to a subset of the Rust language was presented, and the approach is also able to synthesise other functional programming languages like Elm. While Rust is being introduced into the Linux Kernel, making this synthesis option relevant, there does not appear to be a verified Rust compiler. Therefore, for kernel space applications, the combination of CertiCoq and CompCert still seems to be the most viable approach.

The Œuf [42] [126] compiler is able to generate assembly code from a subset of Gallina terms, using CompCert to generate the assembly code. The Gallina functions are reflected to Œuf source code, which includes a guarantee of equivalence using computational denotation. It has also been shown that the resulting assembly code behaves equivalently to the original Gallina term. However, the compiler only accepts a subset of Gallina, i.e. dependent types are not supported, and this limitation does not seem to apply to CertiCoq [9]. Moreover, the set of supported types is limited, e.g. the language does not support function types as arguments of type constructors. Also, the runtime required to compile Gallina terms to assembly is not verified, whereas the minimal runtime of CertiCoq (i.e. the garbage collector) has been verified using CertiGraph. Nevertheless, the runtime is quite small, and it might be possible to adapt it for use in kernel space. But according to the GitHub repository [126], there have been no changes to Œuf since 2019, and the last supported version is Coq 8.5.

Of course, there are other at least partially verified compilers. Some of them do not have Gallina as source language or generate target languages that require comparable effort to make them usable in kernel space, which makes them unsuitable for the use case of this thesis. For example, both Pilsner [43] and Lambda Tamer [13, 14] use an ML-like language as an input language and produce an assembly like language using CPS as an intermediate language. The approach presented in [57] compiles a C-like language (Cito) into assembly, while the compiler itself is formalised in Coq. For the language of Isabelle/HOL [45], a verified synthesis to CakeML [34] does exist [27]. Also, CakeML can be compiled to machine code by a verified compiler backend [56], which is also utilised by other projects such as PureCake [28]. While this provides a high level of trust, kernel space drivers still need to communicate with the low-level functionality of the Linux Kernel, which includes calls to C functions. In principle, this could be done using CakeML's foreign function interface. But the implementation of the foreign function interface contains libc functionality [141, basis/basis_ffi.c], which would have to be adapted for kernel space. Nevertheless, it would be an alternative approach.

3.2.4 Combining Generated and Plain Code

When combining synthesised code with preexisting code, there are aspects to consider in order to avoid unnecessary development effort. That is, the synthesised code should always remain unchanged. There are two main reasons for this. The first is that it would be hard to argue that the synthesised code is still correct after modifications. The second reason is more from the point of view of software engineering. Synthesised code has a life cycle just like any other code. Whenever the formalisation from which the code was extracted changes, another extraction will produce potentially significantly different code. For instance, both the Coq extraction plugin and CertiCoq write synthesised code to files specified in the Vernacular files. Any manual changes made to these files are thus automatically overwritten by the synthesis. So any necessary modifications to that code would have to be stored in additional files, e.g. patch files. But even patches are not necessarily applicable, e.g. if a previously defined function that needs to be modified does not exist in the new synthesis. Modifying the synthesised code can therefore have a serious impact on the maintainability of the whole software project, and synthesised code should always be considered as "drop-in" code, accessed only through well-defined interfaces.

Chapter 4

Discussion

In Section 2.1, the differences between ordinary software (user space software) and operating system components (kernel space software) were discussed. It was shown that there are very different constraints on the two types of software. It was also discussed in Section 3.1 that it is possible to verify existing software, both user space and kernel space functionality with different methods. In Section 3.2, different approaches of synthesising formally correct software were discussed. In order to outline the main differences between verification and synthesis, see Figure 4.1.

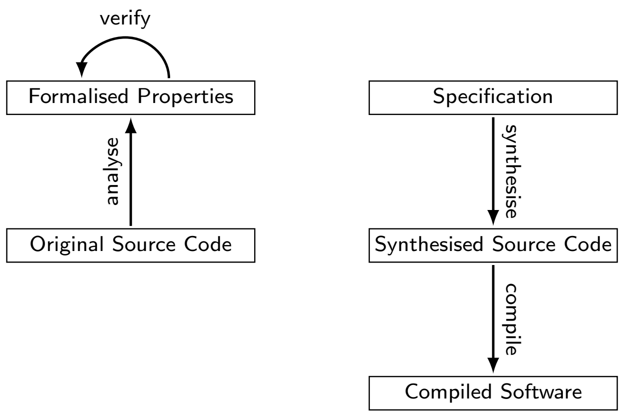

Figure 4.1: Comparing the process of verification and synthesis.

When only the source code is available, the source code needs to be analysed and the properties that should hold for the software must be defined and proved in Coq. This can be done manually or with tool support. However, if the original source code is to be adapted or extended, this involves changes to the original source code, the formalised properties and the verification. Depending on the complexity of the changes to the source code, the required changes to the formalised properties and verification may be more or less extensive.

The situation is different if there is a formal specification from which a functional model can be derived. If the specification changes, the functional model can be derived again as long as the specification is constructive. And if there is a functional model and a formal specification, together with a proof that the former satisfies the latter, then extensions and changes to the functional model only require changes to these components. And that is a clear advantage. In both cases, the functional model can be used to generate source code, which can then be compiled. And, as mentioned earlier, this is possible in a certified process.

In both the verification and synthesis processes, it is important to consider that device vendors typically do not publish a freely available specification of how the driver should behave. In the best case, there is an open source driver implementation that can be analysed. And in worst case, open source drivers are created by reverse engineering existing binary drivers [95]. Therefore, verification of the source code and functional model may not cover all the necessary aspects.

As pointed out in the Subsections 3.2.1 and 3.2.2, it is possible to synthesise OCaml code and Clight code from Coq formalisations using the Coq extraction plugin and CertiCoq respectively. This code is of course usable for user space software, including user space drivers. In fact, the author of this thesis has shown that it is possible to compile synthesised OCaml code into a user space driver for the L4Re Operating Systems Framework using a generalised cross compilation approach [132] as part of the VerSeCloud [21] project.

Furthermore, it was discussed in Subsection 3.2.2 that the synthesis by CertiCoq is verified, unlike that of the Coq Extraction plugin. In addition, using CompCert to compile the Clight code extends the correctness guarantee to the assembly layer. Thus, the verified OCaml extraction for Coq [20] would definitely be more appropriate than the Coq extraction plugin for user space device drivers. However, it was not available at the time of implementing the extensions to the Coq extraction plugin. Furthermore, the verified extraction also lacks the synthesis of foreign function calls, and the results of the extensions to the Coq extraction plugin can be applied as part of future work. Since the synthesised OCaml code tends to have better runtime performance when compiled natively than the Clight code when compiled with CompCert, the Coq extraction plugin (or in the future the verified OCaml extraction) may still be a viable solution for user space drivers.

It has also been pointed out that the code generated by the Coq extraction plugin will not be usable as a kernel space component without significant changes to the runtimes, and this also applies to the verified extraction to OCaml. While CertiCoq also has a runtime, it is significantly smaller than that of OCaml, for example.

Concluding, there is a consensus that it is possible to synthesise certified user space software, but it does not seem to have been addressed whether synthesised code can be used as a Linux kernel space component. And improving the reliability of Linux is desirable, since this operating system is undoubtedly in widespread use[1] although not being verified (unlike seL4 and CertiKOS).

Therefore, the possibility of extending the Coq extraction plugin with foreign function call synthesis and the usability of CertiCoq for the synthesis of kernel space software will be evaluated in the following. It will be shown that even the source code produced by CertiCoq cannot be used for kernel space without modifying either the source code itself or necessary runtime components of CertiCoq. But it will be shown that the necessary modifications can be applied with moderate effort and can be justified in such a way that their correctness can be proven. Also, it will be shown that the same correctness guarantees apply to synthesised kernel space software when using CertiCoq and CompCert.

[1]Although users may not be conscious about that fact.

Part II

Synthesis and Runtimes for Device Drivers

Chapter 5

Extending the Coq Extraction Plugin

For the scope of driver synthesis, it is necessary to communicate with plain C functionality, and there are two ways to communicate between C and OCaml.

The first direction of interest is calling C code from OCaml, which will be covered in Section 5.1. The necessity of calling C functions arises from the fact that low-level functionality, like writing a value to a register, is usually defined in a C library. While it is possible to implement these operations in OCaml, e.g. by using inline assembly code [59], doing so would be like reinventing the wheel.

The other direction of interest is calling synthesised OCaml code from C and will be discussed in Section 5.2. This direction is particularly important when replacing existing device driver code with synthesised code.

5.1 Synthesis of Foreign Function Calls to C

The OCaml language supports foreign function calls to C [89] by declaring an OCaml term to be an external function with a specific name and type. For example, the OCaml addition can be redirected to a C addition function by defining `external add : int -> int -> int = "add_c"`. The function `add_c` then has to be defined as a C function and can be linked with the OCaml code. But the Coq extraction plugin does not support the extraction of such `external` definitions, but only the extraction of axiomatic functions to plain OCaml terms, i.e. `let` expressions.

As an example, consider the Coq script in Listing 5.1. The script defines a function `write_val` that writes a value into a given register, ensuring that the last bit of the value is set by applying a bitwise `or` with 1. The write operation itself is performed by an axiomatic function `outb`.

```
 1  Require Extraction.
 2  Require Coq.extraction.ExtrOcamlNatInt.
 3
 4  (* outb val reg_id *)
 5  Axiom outb : nat → nat → unit.
 6
 7  Definition set_last_bit (val : nat) := Nat.lor val 1.
 8  Definition write_val (val reg : nat) :=
 9            outb (set_last_bit val) reg.
10
11  Extract Inlined Constant Nat.lor ⇒ "(lor)".
12  Extract         Constant outb    ⇒ "C_bindings.outb".
13  Extraction Inline set_last_bit.
14
15  Recursive Extraction write_val.
```

Listing 5.1: Coq code for writing $n|1$ to a register.

The extraction performs some general transformations, i.e. extracting the type `nat` as the OCaml type `int` (line 2), the bitwise function `Nat.lor` to the OCaml specific operation (line 10) and inlining `set_last_bit` (line 12). Additionally, the axiomatic function `outb` is extracted to an OCaml function `outb` from the `C_bindings` module (line 11). A recursive extraction on `write_val` then produces the OCaml code shown in Listing 5.2. It is also possible to inline the definition of `outb`. However, this is omitted for comprehensiveness.

```
(** val outb : int -> int -> unit **)
let outb = C_bindings.outb

(** val write_val : int -> int -> unit **)
let write_val val0 reg =
  outb ((lor) val0 (Stdlib.Int.succ 0)) reg
```

Listing 5.2: OCaml code synthesised from `write_val`.

The advantage of extraction is obvious; the correctness of the extracted code can be at least partially assumed, as pointed out in Subsection 3.2.1. However, in order to successfully compile the program, the implementation of the `C_bindings.outb` function must still be provided by adding a file `c_bindings.ml`. That is, the definition `external outb : int -> int -> unit = "outb_c"`[1] is required, which states that the evaluation of `outb` is done by calling an external

[1]The Linux `outb` implementation is a void function returning nothing.

function `outb_c`. The C function itself can be defined in a C file as shown in Listing 5.3. This function is called by OCaml and transforms the given `value` parameters to the correct C types, i.e. unsigned 8-bit integers, performs the intended functionality and then returns a `value` to OCaml, in this case the `unit` type. It is important to note that a type mismatch between `C_bindings.outb` and the C function `outb_c` cannot be detected. This is due to the fact that OCaml propagates the type `value` for parameters and return values. Thus, the `ocamlopt` compiler will not produce any compilation errors. It is therefore up to the developer to ensure that the parameters are correctly interpreted on the C side. However, it is possible to omit the type transformation by passing unboxed values [89, Sec. 11] which will also be discussed in Section 5.3.

```
CAMLprim value
outb_c (value o_val, value o_reg_id){
  // Interpret reg_val and reg_id as unsigned integers
  uint8_t reg_val = (uint8_t)Unsigned_int_val(o_val);
  uint8_t reg_id  = (uint8_t)Unsigned_int_val(o_reg_id);
  // call the C function implementation
  return (Val_unit);
}
```

Listing 5.3: C wrapper function for `outb_c`.

But the workflow of extracting an axiomatic function as an OCaml function and then redirecting it to an external C function in an additional module is quite cumbersome. Problems also arise if the type of the axiomatic function is changed in Coq for some reason. For example, if the extraction is changed from type `nat` to `int` or vice versa, the developer needs to adapt both the C wrapper function and the `C_bindings` module. That is, if lines 2 and 11 in Listing 5.1 are removed, compilation will produce the type mismatch error shown in Listing 5.4. While the adjustment of the wrapper function cannot be easily avoided, the modification of the external function declaration can be solved automatically. Especially since the target type of the extraction is defined internally in Coq.

```
File "write_val.ml", line 87, characters 7-24:
87 |   outb (lor0 val0 (S O)) reg
            ^^^^^^^^^^^^^^^^^
Error: This expression has type nat but an expression was expected
of type
         int
```

Listing 5.4: Compilation error after changing types of `outb`.

It is therefore clearly more desirable to be able to synthesise the external declaration during Coq extraction. This functionality has been implemented for OCaml extraction by the author of this document in the following way. A new Coq command **`Extract Foreign Constant`** `@qualid ⇒ @string`, which is similar to the existing **`Extract Constant`** command has been introduced. While the original command marks the axiomatic function referenced by `@qualid` as a (potentially inlined) `custom` term, the new command marks it as `foreign custom` term with the name `@string`.

During extraction, both cases are handled by the same implementation. In general, the Coq extraction plugin synthesises an `mli` file containing the OCaml term declarations, i.e. the interface, and an `ml` file containing the term definitions, i.e. the implementation. For both **`Extract Constant`** and **`Extract Foreign Constant`**, a `val` declaration is synthesised in the output `mli` file. When synthesising the `ml` file, the extraction plugin then distinguishes between `foreign custom` and `custom` terms. While `custom` terms are synthesised as regular OCaml terms (i.e. `let` expressions) as before, `foreign custom` terms are synthesised as `external` expressions including the type. Thus, when replacing Line 12 in Listing 5.1 with **`Extract Foreign Constant`** `outb ⇒ "outb_c".`, the result of the extraction is as shown in Listing 5.5.

```
(** val outb : int -> int -> unit **)
external outb: int -> int -> unit = "outb_c"

(** val write_val : int -> int -> unit **)
let write_val val0 reg =
  outb ((lor) val0 (Stdlib.Int.succ 0)) reg
```

Listing 5.5: OCaml code synthesised from `write_val` with extended synthesis.

The obvious advantage here is that the necessary changes in the Coq script are minimal, i.e. using foreign extraction instead of normal extraction. In addition, the type-correctness of `outb` is guaranteed, which eliminates potential sources of compilation errors. The changes to the Coq extraction plugin described here have been submitted as a Pull Request for inclusion in Coq [130] and will be part of Coq 8.20. They also include the possibility to display the currently defined foreign functions with the command **`Print Extraction Foreign`**. In the following, the synthesis of calls from C to OCaml is discussed.

5.2 Synthesis of OCaml Entry Points

In an iterative process of replacing unverified C code with synthesised OCaml code, it is necessary to be able to call the synthesised code. OCaml has the functionality to expose functions by global names as closures and make them callable from C code [89, Sec. 7]. Registering an OCaml function, e.g. `compute_val` with an `int` parameter, as a callable closure is done with the command

`let _ = Callback.register "compute_val" compute_val.` [2]

On the C side, the OCaml runtime must first be initialised. Then the closure can be searched with `caml_named_value("compute_val")`, which returns the closure as `value *` variable, e.g. `clo`.

This closure can then be called with the `caml_callback(*clo, Val_int(n)))` function, which evaluates the closure and returns its return value. It is clearly possible and not much effort to define the callback registrations for each synthesised function to be exposed manually. However, when refactoring Coq formalisations, for example, it is not uncommon for the names of some functions to change, making manual adjustments to the callback registration necessary. Furthermore, in order not to modify the synthesised OCaml files, all callback registrations must be stored in an additional file.

This effort can be avoided because the Coq script also contains the information about the name of the extracted function. Therefore, the Coq extraction plugin was extended to also synthesise closure registration calls. This was done by introducing the Coq command **`Extract Callback`** `@string_opt @qualid`. This command marks the specified `@qualid` with the optional name given as `@string_opt` as a callback in an internal map. During extraction, whenever the OCaml code for a function is synthesised, the implementation checks whether it is a callback function and adds the callback registration call after the function definition. As an example, consider Listing 5.1, which uses `write_val` from Listing 5.6. Here a function is defined that calls `write_val` with a fixed register value and is then declared as a callback function accessible by the global name `write_reg`.

```
Definition write_pcspkr_reg (val : nat) := write_val val 0x61.
Extract Callback "write_reg" write_pcspkr_reg.
Recursive Extraction write_pcspkr_reg.
```

Listing 5.6: Coq code for writing $n|1$ to register `0x61`.

The new extraction scheme then produces the result shown in Listing 5.7. The call to `Callback.register` (line 12) is fully qualified according to the OCaml

[2]The expression "`let _`" can be used instead of "`let ()`" and the global name can be arbitrary but must be unique.

standard library to ensure that no naming conflict when a formalisation includes a function `Callback.register`[3].

```
(** val outb : int -> int -> unit **)
external outb: int -> int -> unit = "outb_c"

(** val write_val : int -> int -> unit **)
let write_val val0 reg =
  outb ((lor) val0 (Stdlib.Int.succ 0)) reg

(** val write_pcspkr_reg : int -> unit **)
let write_pcspkr_reg val0 =
  write_val val0 (Stdlib.Int.succ [...] )

let () = Stdlib.Callback.register "write_reg" write_pcspkr_reg
```

Listing 5.7: Synthesised code for writing $n|1$ to register `0x61`.

As shown, these extensions of the Coq extraction plugin do have a positive impact on the overall development process and maintainability. The internal name of the registered callback can be automatically updated while preserving the global name, which eliminates the possibility of compilation errors when changing the function name in the Coq formalisation. It is also easy to add and remove exposed OCaml functions in the proof script. Finally, no additional files containing the callback registrations are required. However, the synthesis of the callback registration does not have any impact on the correctness of the complete software at runtime. The modifications to the Coq extraction plugin described here have also been submitted as part of the Pull Request [130] for inclusion in Coq and will be part of Coq 8.20. They also include the possibility to display the currently defined callbacks with the command **`Print Extraction Callback`** and to reset them with **`Reset Extraction Callback`**.

5.3 Open Work

While the extensions presented can be considered as improvements, they can and should also be seen as a starting point for further extensions. As noted, the extraction of foreign function calls and callbacks is limited to OCaml. But the Coq extraction plugin is also able to synthesise Haskell and Scheme. While there are syntactical differences between the OCaml, Haskell and Scheme FFI, the concept

[3]This was a review suggestion given by Gaëtan Gilbert (Inria).

for OCaml extraction can in principle be applied for extracting the other languages, too.

There are even more possible extensions for OCaml extraction. The most important are performance-critical optimisations supported by the OCaml language [89, Sec. 11] when compiling code natively, i.e. with `ocamlopt`. As described in Section 5.1, the C-side functions receive parameters as `value` types and must transform the values into the correct C types, i.e. *unbox* them. Return values must also be *boxed* to the `value` type. But OCaml has the functionality to work with *unboxed* values if the OCaml types are `float`, `int32`, `int64` or `nativeint` [89, Sec. 11.1]. In this case, the types can be transformed to the C types `double`, `int32_t`, `int64_t` or `intnat`, respectively. According to the documentation, the following changes are required if, for example, the function `outb` from Section 5.1 is called with parameters of type `int64`. Instead of defining the external by `external outb: int64 -> int64 -> unit = "outb_c"`, the `int64` types can be annotated, i.e. written as `(int64 [@unboxed])`, while the function body can then be defined as `"outb_c_wrap" "outb_c"`. This tells the OCaml compiler to call the C function `outb_c_wrap`, which then calls `outb_c` while unboxing the input parameters. On the C side, the functions `outb_c_wrap` and `outb_c` then need to be defined as shown in Listing 5.8.

```c
void outb_c (int64_t val, int64_t reg_id) {
  // check if val and reg_id are <= max_uint8
  // write the val to reg_id.
  return;
}

CAMLprim value outb_c_wrap (value o_val, value o_reg_id) {
  // call outb_c with reg_val and reg_id as int64_t
  outb_c(Int64_val(o_val), Int64_val(o_reg_id));
  return (Val_unit);
}
```

Listing 5.8: C wrapper functions for `outb_c_wrap` and `outb_c`.

The advantage here is that `outb_c_wrap` is rather generic. In fact, this function could even be synthesised by Coq, although this is not a trivial task. While `floats` are defined in the Coq standard library, there seem to be no definitions for `int32` and `int64`. Instead, there is a 63-bit definition of integers. However, if a Coq formalisation contains proofs that certain integral (or `nat`) values supplied to an axiomatic function are in the range of `int64`, extracting them `unboxed` would be type-safe.

Under this assumption, the Coq extraction plugin could be extended by a command **`Extract Foreign Constant Unboxing`** `@qualid ⇒ @string`. This command would have to mark the `@qualid` as a `foreign unboxing custom`, for example. During extraction, the type of such a function must be analysed in detail. Whenever the target `type` segment is `float` or `int`, the output type segment for the external definition should be `(float [@unboxed])` or `(int64 [@unboxed])` respectively. For the `int64` case, any function that uses the axiomatic function must be synthesised to use the transformation of an `int` parameter `p` to `int64` by applying `StdLib.Int64.of_int p`. Additionally, the extraction could synthesise C wrappers, i.e. for the given `@string`, a function `@string_wrap` could be generated as shown in Listing 5.8. This way, the `@string_wrap` function would be type-safe. Additionally, the compiler would then be able to type-check the call to `outb`, which would increase the type-safety for integral or floating-point parameters and return values.

Another interesting concept that can be used when compiling OCaml natively is the use of the `[@@noalloc]` annotation [89, Sec. 11.2]. This annotation signals to the compiler that the external function will not allocate memory, e.g. by `malloc`, nor will it raise exceptions or release the “master lock” [89, Sec. 12.2]. This is especially true for register write operations, which are common in driver development. Annotating the function accordingly has the effect that the call to the function is not wrapped in the `caml_c_call` function, which would normally apply garbage collection. In this way, the overhead of garbage collection can be omitted. [89, Sec. 11.2] Extending the OCaml extraction plugin in this case is much simpler than for *unboxing*. It is sufficient to define a Coq function **`Extract Foreign Constant Noalloc`** `@qualid ⇒ @string` which marks the `@qualid` as a `foreign noalloc custom`. For instance, when synthesising `outb` with the **`Noalloc`** extension, the extraction would then have to synthesise

```
external outb: int -> int -> unit = "outb_c" [@@noalloc].
```

But additionally, the Coq command should emit a warning to to the user, that it is necessary to check whether the function really fulfils the preconditions.

While it is possible to reset the callbacks as described in Section 5.2, it is not possible to do the same for foreign functions. In fact, this is not even possible for axiomatic extraction to OCaml functions with the **`Extract Constant`** [130] command. While there is a **`Reset Extraction Inline`** command, it merely removes the information that the axiomatic functions should be inline. The reason for this is that when marking an axiomatic function as `custom`, a substitution from `@qualid` to `@string` is stored internally in Coq in a “superglobal_object”. This means that the substitution applies whenever the module in which it was defined is required by another module. It is therefore not trivial to remove the substitution. This issue has raised by the author in the Pull Request [130].

Chapter 6

Extracting Certified Device Driver Code to Clight

In the last chapter, the extraction of formalised and verified software to OCaml source code was discussed. But as described in Subsection 3.2.2, it is possible to synthesise Clight code using CertiCoq. While the Clight code generated from the formalisation by CertiCoq can be used without modification in both user and kernel space, this is not the case for the additional required files. As described in Subsection 3.2.2, CertiCoq provides a set of files that are needed to compile the generated source. These are the generated glue code, some headers (`values.h`, `m.h`, `config.h`) and additionally, and most importantly, the garbage collection. For the scope of this document, the stack-based garbage collection (`gc_stack`) will be considered. Also, if the provided primitive files (e.g. `prim_int63`) shall be used, these are also required. For compiling user space device drivers, no modifications are required, as a libc is usable. However, for compiling kernel modules, changes are required, as implied in Section 2.1.

In the following, the transformation of the CertiCoq runtime and changes to the glue code files are discussed, starting with the most complex aspect, i.e. the garbage collection.

6.1 Adapting the CertiCoq Garbage Collection

Since the garbage collection (GC) has to run in kernel space, it is necessary to get rid of the loss of control flow caused by calls to `exit` and `assert`. As pointed out in Section 2.1, these functions should not, and partially even cannot be used in kernel space. To solve these issues, the garbage collection should react lazily on failure, i.e. the GC has to be transformed so that errors are propagated to the public functions. This way, the errors can be handled by the calling code.

The garbage collection exposes 8 functions (by making them accessible in the header `gc_stack.h` [142]), as shown in Listing 6.1. In the following, the original source code [142, gc_stack.c] is discussed. The `make_tinfo` function allocates heap memory for the execution of the synthesised code and returns a `thread_info` structure containing the closure (the synthesised function to be called), its parameters and its return value. The `garbage_collect` function removes unused data from previous computations and is called by the synthesised code itself. The `free_heap` function frees the heap memory allocated by `make_tinfo`, while `reset_heap` only clears the heap without freeing it. The functions `export_heap` and `extract_answer` can be used to copy the closure return value and the complete heap respectively. These functions are not necessarily used by synthesised code. Also, the stack-based garbage collection implementation file does not contain an implementation of `extract_answer`. The `certicoq_modify` function applies a mutable write barrier to the `thread_info`, and finally the `print_heapsize` function does exactly what its name implies - it prints the allocated, used and remembered heap size.

```c
struct thread_info *make_tinfo(void);
void garbage_collect(struct thread_info *ti);
void free_heap(struct heap *h); // @safe
void reset_heap(struct heap *h); // @safe
value* extract_answer(struct thread_info *ti);
void* export_heap(struct thread_info *ti, value root);
void certicoq_modify(struct thread_info *ti, value *p_cell,
                     value p_val);
void print_heapsize(struct thread_info *ti); // @safe
```

Listing 6.1: Function definitions in the `gc_stack.h` of CertiCoq.

Some of the functions will never fail, i.e. call assertions or `exit`[1]. These functions are `free_heap`, `reset_heap` and `print_heapsize`, as indicated by the `@safe` documentation tag in the listing. The most important functions, `make_tinfo` and `garbage_collect`, may fail. Although `extract_answer` and `export_heap` are not automatically used by the synthesised code, and are not necessary for kernel space drivers, they can fail and the failures should be handled.

When analysing the garbage collector in depth, there is one main observation: All functions that may fail are either functions that return `void`, i.e. nothing, or functions that return a pointer. And this opens up the possibility of a schematic approach to adapting the garbage collection. That is, for each function f that returns `void`, it is possible to construct a function f' that returns the boolean

[1]Precisely, `exit` is called via a wrapper function `abort_with` that additionally prints an error.

value `false` whenever the original function f evaluates to `exit` or `assert`, and `true` otherwise. Furthermore, for any function f that returns a pointer, it is possible to construct a function f' that returns `NULL` (the null pointer) whenever the original function f evaluates to `exit` or `assert`. Returning a null pointer is a viable solution because this value is generally considered to be an invalid value. Also, if f would have returned a null pointer under normal conditions, f' would do the same. Once these modified functions have been constructed, the functions that call them have to be adapted to propagate the errors to the public functions. This schematic approach will be used in the following to transform the garbage collection.

The `certicoq_modify` function has a fairly simple structure, as it only sets a mutable write barrier, does not call any other functions, and only fails with an assertion if `ti->alloc < ti->limit` does not hold, i.e. if the `alloc` pointer reaches or exceeds the limit. This can be easily solved by making the function return boolean values, and making it behave differently depending on whether it is compiled for user space or kernel space[2]. Then the function can still use the assertion in user space, while printing an error message before returning false (0) in kernel space. Also, in both cases, the function has to return true (1) at the end of the function body. In the following, the transformation of the more complex functions is described.

6.1.1 Making make_tinfo Fail Lazily

When the public function `make_tinfo` is called, the private functions `create_heap` and `create_space` are involved in the call graph, as shown in Figure 6.1, and the edges are labelled with the order of execution.

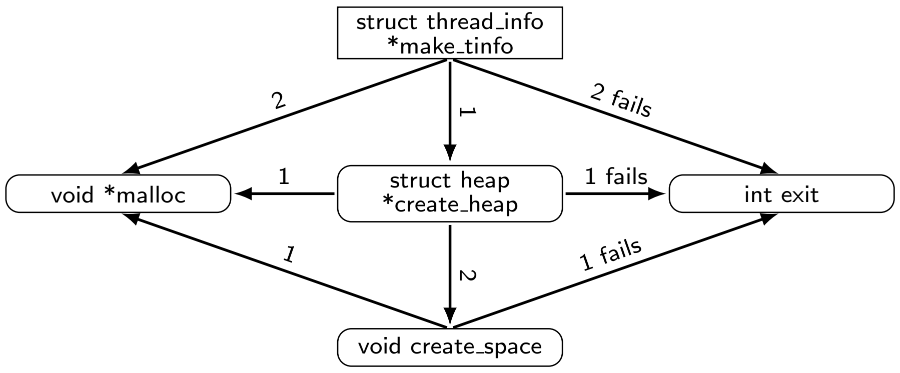

Figure 6.1: The original call graph when calling `make_tinfo`.

[2]Using the compile barrier `#ifdef CERTICOQ_KERNEL_SPACE` ... `#else` ... `#endif`.

Here, `make_tinfo` first calls `create_heap`, allocates the `thread_info` and calls `exit` if the allocation fails. The function `create_heap` itself tries to allocate the heap memory, and calls `exit` on failure. If the memory allocation is successful, `create_space` is called. Then `create_space` attempts to allocate the next generation and calls `exit` on failure.

This functionality can be transformed using a lazily failing approach by starting with `create_space`, i.e. the bottom of the call graph. As can be seen, `create_space` is a void function in the original implementation, i.e. it returns nothing. To catch and propagate failures, the function has to be redefined to return a boolean value, for example. Then, instead of exiting on a failed space allocation, the function can return false, while returning true on success.

Now that `create_space` propagates failures to `create_heap`, the behaviour of `create_heap` can be adapted by returning NULL if either the allocation of the heap returns NULL or the call to `create_space` evaluates to false. Thus, `create_heap` returns NULL if and only if `malloc` or `create_space` fails.

This transformation allows `make_tinfo` to detect a failed call to `create_heap`. Again, NULL can be returned if either `create_heap` returned NULL, or if the allocation of the `thread_info` failed, making the call to `exit` obsolete. The resulting call graph is shown in Figure 6.2.

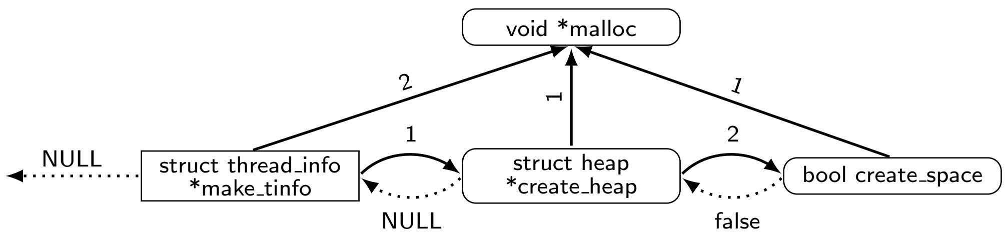

Figure 6.2: The call graph when calling `make_tinfo` with lazy failing.

The advantage of this transformation is that errors propagated by the private functions `create_heap` and `create_space` can also be caught by other public functions.

6.1.2 Making garbage_collect Fail Lazily

When the `garbage_collect` function is called, it in turn calls the private functions `create_space`, `do_generation` and `resume`. Also, `do_generation` calls the functions `forward_remset`, `forward_roots` and `do_scan` which in turn all call the function `forward`, as shown in Figure 6.3. With the changes presented in the last subsection, the functions `create_space` and `forward` are both safe, as indicated by the dotted border. Since `forward_roots` and `do_scan` do not exit or assert and

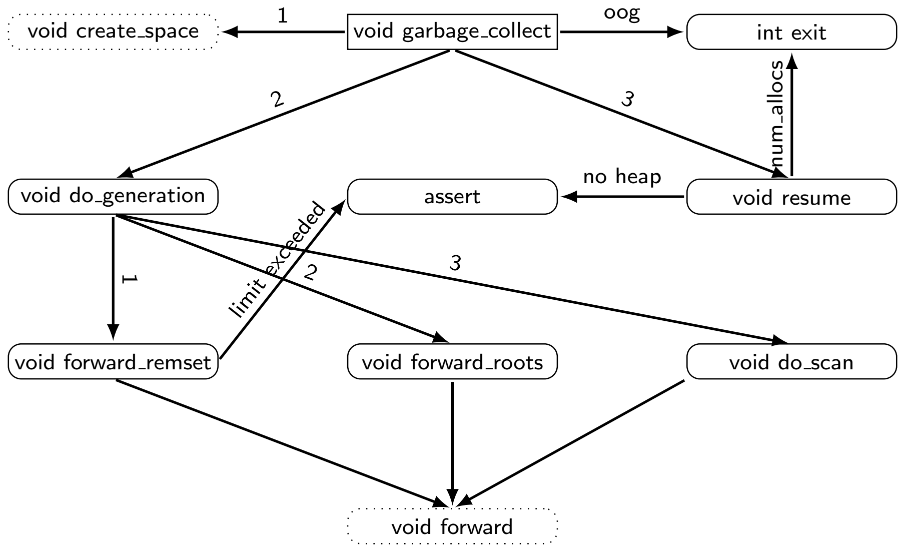

Figure 6.3: The original call graph when calling `garbage_collect`.

use the safe `forward` function, they can remain unchanged. The only functions that do exit or assert are `forward_remset`, `resume` and `garbage_collect` itself. Specifically, `resume` exits if not enough space has been freed and `garbage_collect` exits if the GC has run out of generations. And all these functions are void functions in the original implementation. Thus, their return value can be changed to boolean, as was done for `create_space`. The result of the transformation is shown in Figure 6.4, omitting the functions that did not need to be changed.

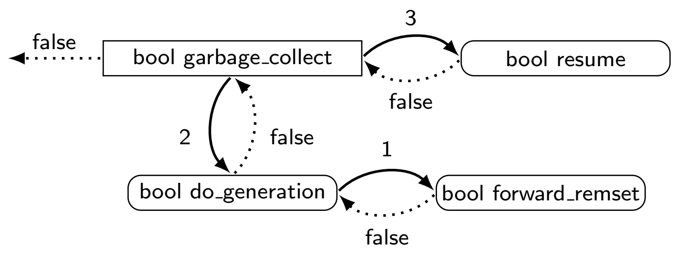

Figure 6.4: The call graph when calling `garbage_collect` with lazy failing.

Here, `resume` returns false if no heap has been allocated (previously assertion) or the nursery is too small. Otherwise it returns true. The `forward_remset` function returns false if the assertion fails, and true otherwise. This return value is then propagated by `do_generation` to `garbage_collect`.

Finally, `garbage_collect` returns false if and only if one of the mentioned functions (or `create_space`) failed or if the GC has run out of generations, i.e. no additional spaces can be created.

6.1.3 Making export_heap Fail Lazily

The public function `export_heap` calls the functions `create_space`, `create_heap`, `garbage_collect_all` and `do_generation`, as shown in Figure 6.5.

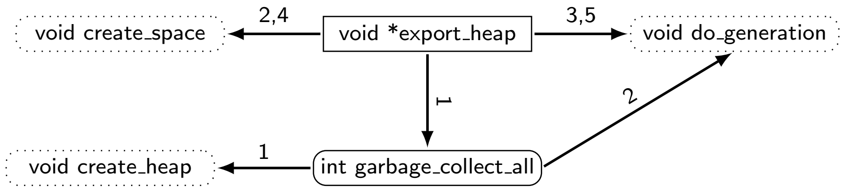

Figure 6.5: The original call graph when calling `export_heap`.

Here the functions `create_space`, `create_heap` and `do_generation` have already been transformed, as indicated by the dotted borders. And indeed, neither `export_heap` nor `garbage_collect_all` do fail by calling exit or assertions. But `garbage_collect_all` does call `create_heap` and `do_generation`. Thus, the function has to be transformed to propagate failures from these functions. Both functions return false on error after transformation. And `garbage_collect_all` returns an integer, which makes propagation more complicated, but not impossible.

The solution here is that `garbage_collect_all` returns a value `i` which is defined by an iteration in the interval $[0, MAX\_SPACES - 2]$[3]. So any value outside this interval is an error value. And since the return value has to be of type int (which is actually `signed int`), a valid error return value is -1. So the function can be redefined to return -1 if and only if `create_heap` or `do_generation` fails.

Then it is possible to catch a failed `garbage_collect_all` in `export_heap` and the function must also catch failed calls to `create_space` and `do_generation` functions. Additionally, there are two memory allocations that create memory for spaces to do the export. These memory allocation calls are not guarded in any way in the original implementation. Thus, a failed allocation would produce a NULL pointer, which must also be handled by returning NULL in this case. In summary, `export_heap` must return NULL if and only if one of the memory allocations or calls to the described functions fails. It is also necessary to ensure that successfully allocated memory is freed when one of the functions fails, and this has been done in the course of the GC transformation.

[3]The loop expression limits `i` to satisfy the condition $i < MAX\_SPACES - 1$.

6.1.4 Considering Purely Private Functions and Failing

After having transformed all public functions and functions called by them to lazily failing variants, there are still private functions that need to be addressed. One of these functions is `uintnat gensize(uintnat words)`, which is actually disabled by guarding it with an "if false" macro[4]. This function computes the size of the next generation and calls exit via `abort_with` if the size exceeds the size of an unsigned integer, i.e. would be too big for the address space. Also, the function asserts that the size of the next generation is at least twice the size of the current one. Assuming that the current generation cannot have a size of 0, the size must be positive. Thus, the size of the next generation must also be positive. And since the return value is `uintnat`, which includes the value 0, this value can be used as the error value. Now the call to the `exit` function can be replaced by returning 0. And similar to the transformed version of `certicoq_modify`, the assertion can be an `assert` for user space and return 0 in kernel space.

Now that all calls to `abort_with` have been removed, the function itself can also be removed. Also, no assertions are called when the code is compiled for kernel space. Consequently, the include to `assert.h` can be guarded by a compile barrier[5] to ensure that it is removed for kernel space.

To ensure that the new garbage collection works as before, it must still fail in user space code. In particular, the call to `garbage_collect` from synthesised code must still exit in user space, but not in kernel space. And since it is the only garbage collector function that is always called by the synthesised code, no other functions need to explicitly call the `exit` function, as invalid values can be caught by the calling plain C code. The `garbage_collect` function in the implementation file[6] can then be renamed to `do_garbage_collect`, for example. The dependent failing can then be implemented as shown in Listing 6.2.

```
// In lazily_failing_gc_stack.h
#ifdef CERTICOQ_KERNEL_SPACE
_Bool
#else
void
#endif
garbage_collect(struct thread_info *ti);

// in lazily_failing_gc_stack.c
#ifdef CERTICOQ_KERNEL_SPACE
```

[4]`#if 0 ... #endif`.

[5]`#ifndef CERTICOQ_KERNEL_SPACE ... #endif`.

[6]`lazily_failing_gc_stack.c`

```
inline _Bool garbage_collect(struct thread_info *ti) {
  return do_garbage_collect(ti);
}
#else
inline void garbage_collect(struct thread_info *ti) {
  if (0==do_garbage_collect(ti))
    exit(1);
}
#endif
```

Listing 6.2: Making `garbage_collect` dependently fail.

In the header file, the return type is made dependent on whether the file is compiled for kernel or user space. In the implementation file, the new implementation of the `garbage_collect` function then returns the result of the `do_garbage_collect` function in kernel space, and exits in user space if the return value of the function is 0, i.e. false. With all these changes, the garbage collection is still compatible with user space applications. While lazy failing introduces a latency before the eventual call to the `exit` function, this latency is not relevant in user space, since both the original and the modified garbage collection will fail in user space and stop the process as a result. Also, the lazily failing GC can be a basis for other UNIX operating systems (e.g. BSD) where the same restrictions on the use of `exit` and `assert` apply.

6.1.5 Specialising for Linux Kernel Modules

Once the garbage collector implementation has been transformed, it needs to be specialised for kernel module applications, preferably in new files. As described in Section 2.1, the functions `printf`, `fprintf`, `malloc` and `free` cannot be used. Therefore, the specialised implementation has to use functions provided by the Linux Kernel headers.

The easiest aspect to solve is the printing functionality. All calls to `printf` and `fprintf` (if the output is `stderr`) can be replaced by calls to `printk`. Specifically, all informational printing instructions are set to log level `KERN_INFO`, and all error messages, e.g. the ones introduced in the transformation into a lazily failing GC, are set to log level `KERN_ERR`. Since `printk` does not support floating-point conversion specifiers, the occupancy logging output in `do_generation` must print floating-point values as integral values, which clearly reduces precision. Furthermore, as described in Section 2.1, there is no alternative to `fprintf` in kernel space, and reading or writing files from kernel modules is discouraged. Thus, the `printtree` function effectively cannot be used, and the `in_heap` function is never called. While these functions are only compiled when the `DEBUG` flag is set, they should generally be ignored when compiling for kernel space. This is done by compiling them only if `CERTICOQ_KERNEL_SPACE` is not defined. Additionally, the dependent include of `assert.h` and all assertions and user space specific implementations can be removed.

Also, kernel modules are compiled with many warnings enabled, like `-Werror=strict-prototypes`. This option causes a warning to be issued when a function declaration or definition does not contain argument types. If a function has no arguments, as is the case with `create_heap`, then the argument type of the function must be `void`. And in this case, this warning is even treated as an error. It is of course possible to override this warning, but from an engineering point of view, garbage collection code should satisfy the same constraints as any other kernel module component.

What remains is the adaption of memory allocations to the Linux Kernel. As pointed out in Section 2.1, kernel space software must use kernel-specific functions to allocate and free heap memory. But so far, the garbage collection still uses `malloc` and `free` from the user space standard library. The garbage collection allocates memory in the functions `create_heap`, `create_space`, `make_tinfo` and `export_heap`. It is important to note that `create_space` actually allocates an array of *n* cells. Thus, it is possible to use `kmalloc_array` in `create_space` and `kzalloc` in the remaining functions. One hurdle is that the kernel-specific functions are defined in `linux/slab.h`. And this header file transitively imports the header `thread_info.h`, which contains a definition of - as can be expected - a `thread_info` structure. Thus, when including `linux/slab.h` in the garbage collection instead of `stdlib.h`, the garbage collection compilation will eventually fail because of the redefinition. But this problem can be solved by defining a new memory allocation implementation file, as shown in Listing 6.3.

```c
#include "certicoq_alloc.h"
#include <linux/slab.h> // for kzalloc, kmalloc_array and kfree.

void * alloc(size_t sz)
{ return kzalloc(sz, GFP_KERNEL); }

void * alloc_array(size_t n, size_t sz)
{ return kmalloc_array(n, sz, GFP_KERNEL); }

void free(const void *objp)
{ kfree(objp); }
```

Listing 6.3: A custom memory allocation compilation unit, `certicoq_alloc.c`.

Here, the include of `slab.h` is done in the additional C file, which is compiled as a separate compilation unit. And in this file, no redefinition of `thread_info` will occur. In fact, the `thread_info` definition from the Linux Kernel header is not even used, and is thus eliminated. When exposing the functions `alloc`, `alloc_array` and `free` in the header file for the new compilation unit, and including the file in garbage collection instead of `stdlib.h`, the new functions can be used. That is, `alloc` is used in `create_heap`, `make_tinfo` and `export_heap`, while `alloc_array` is used in `create_space`.

A side effect of not using the `stdlib.h` is that the `stdint.h`, which contains definitions of integral types (e.g. `size_t`), is not included. But these definitions are needed for memory allocation. This can also be easily solved by including `linux/types.h` in the memory allocation header file.

While it is possible to free or reset the heap, there is no explicit function to deallocate the `thread_info`. Although it is possible to do this manually by first calling `free_heap` and then freeing the `thread_info` itself, it is easy to forget to free the heap. Thus, a public function `free_tinfo` has been implemented to simplify this task.

Now the garbage collection is fully compliant for use in Linux kernel space. However, there are still other files required during compilation that need to be adapted. In the following section, all changes to the original CertiCoq files or synthesised files are discussed.

6.2 Adapting the Synthesised Code and Other Runtime Components

As described in Subsection 3.2.2, CertiCoq synthesises Clight code representing the formalised functionality and the result of the synthesis is a C and header file, e.g. `prog.c` and `prog.h`. Also, CertiCoq is able to synthesise glue code producing the files `glue.c` and `glue.h`, for example. The necessary changes to these files or the files they include are described below.

6.2.1 Adapting the Synthesised Code

The files containing the synthesised functionality do not directly use any libc functionality. The only headers being included by the code are `prim_floats.h`, `prim_int63.h`, `coq_c_ffi.h` and `gc_stack.h`, which provide primitive operations for floating-point and integer arithmetic, communication with Coq and the garbage collection respectively. All these files are part of the CertiCoq runtime [142]. However, the includes of the primitives and the Coq FFI files can be omitted as described in Subsection 3.2.2. Also, the synthesised code can use the GC that was specialised for kernel space in the last section instead of the original one. Concluding, if the default includes are omitted, no changes are required to the synthesised code itself.

Communication with Coq is not necessary for kernel modules, and this include is definitely superfluous. However, the primitives files may be useful and necessary if primitives are used by the synthesised code, and these files actually use libc functionality. For example, the primitive integer operations implementation uses printing functionality (from `stdio.h`) and also includes both `stdlib.h` and `gc_stack.h`. Again, while the specialised GC can be used safely, the other includes must be addressed separately. The printing functionality is used via a `trace` macro, which expands to `printf` if `CERTICOQ_TRACE` is defined, and otherwise to a comment.

```
#ifdef CLIGHT_KERNEL_CODE
  #include <linux/printk.h>
  #ifdef CERTICOQ_TRACE
    #define trace(...) printk(KERN_INFO __VA_ARGS__)
  #else
    #define trace(...) //printk(KERN_INFO __VA_ARGS__)
  #endif
#else
  #include <stdio.h>
  #ifdef CERTICOQ_TRACE
    #define trace(...) printf(__VA_ARGS__)
  #else
    #define trace(...) //printf(__VA_ARGS__)
  #endif
#endif
```

Listing 6.4: A custom `stdio` wrapper header.

This can be easily solved by defining a wrapper file, e.g. `certicoq_stdio.h` as shown in Listing 6.4. Here, the new file includes `linux/printk.h` for kernel space and `stdio.h` for user space, and redirects `trace` to the appropriate implementation. Then, when including this file instead of the original, printing will work transparently for both user and kernel space. The include to `stdlib.h` done by the primitive file is not even used by the file itself. Also, the include to the GC itself already includes memory allocations, and the include to `stdlib.h` is superfluous and can be removed[7].

The implementation of primitive floating-point operations uses the primitive integer operations provided by the file `prim_int63.h`, and additionally includes

[7]This was confirmed by Matthieu Sozeau in private communication.

the stdint.h (for integral type definitions) and the math.h. Both the stdint.h and math.h includes can be replaced by includes of certicoq_stdint.h and certicoq_math.h. The file certicoq_stdint.h will then include linux/types.h for kernel space and stdint.h for user space, and the header file certicoq_math.h will include either linux/math.h or math.h. However, the math library should be used with care, as discussed in Section 2.1.

6.2.2 Adapting the Glue Code

As noted in Subsection 3.2.2, the glue code includes functionality to construct Coq types (such as nat or Z) with C functions. These functions do not use any libc functionality and can be used as components of a kernel module without modification. But the files also contain functions for printing Coq types to the console. Here again, the use of stdio.h breaks the usability of the Clight code in kernel modules. In principle, it would be possible to use the certicoq_stdio.h described in the previous subsection and extend it with wrapper functions for printf that redirect to printk in kernel space. However, this would not give the expected result. The reason is that printing is done recursively on the structure of the given type. For example, when printing a value of type nat with the function print_Coq_Init_Datatypes_nat, the function processes the value recursively, as shown in Listing 6.5.

```
printf(lparen_lit);
printf(*(names_of_Coq_Init_Datatypes_nat + $tag));
printf(space_lit);
print_Coq_Init_Datatypes_nat(*((value *) $args + 0));
printf(rparen_lit);
```

Listing 6.5: Extract from nat printing functionality.

When redirecting printf to printk, each parenthesis, S and O would be printed on a separate line, as the default log level (KERN_INFO) is defined this way. It is possible to use the continuation log level (KERN_CONT) here, which will give the expected result. But as described in Section 2.1, this log level may not be secure.

So there are three possible solutions. The simplest is to remove the printing functionality, as it is only used for debugging purposes and is not necessary for compilation[8]. Also, removing the functionality does not affect the correctness of the remaining code, as long as it is not used by the synthesised code. A better solution here would be to provide a variant of the glue code generation for CertiCoq that omits the printing functionality, but this approach is beyond the scope of this

[8]This is the option taken for the example driver implementation.

work. The most elegant solution would be to split the printing into two stages. The first stage (string_of_nat) should construct a string, represented as char *. Then the function print_Coq_Init_Datatypes_nat could just print the generated string. In principle, the string could be constructed using the strcat or strncat functions, which are supported for both user space (string.h) and kernel space (linux/string.h), but are considered unsafe [69]. Instead, the documentation recommends using the scnprintf function. But to use this function, the size of the string to be produced must be known (i.e. computed) a priori. And this computation introduces additional overhead. Although this would be the best and most portable solution, it would require significant changes to CertiCoq that are beyond the scope of this thesis.

With these changes, the glue code for constructing and destructing Coq types can be used in kernel space. But when communicating between C and synthesised Clight code, the use of raw glue code functions can be quite cumbersome. This can be solved using primitives, and is discussed in the following subsection.

6.2.3 Adapting Types, Transformations and Primitives

When communicating between synthesised Clight code and plain C code, all parameters and return values must be of type value. This and other primitive types are defined in CertiCoq's values.h, which is intended to be compatible with the definitions in OCaml's mlvalues.h [123], as noted in Subsection 3.2.2. And, as noted in Section 2.2, kernel modules are compiled with many parameters, making it necessary to adapt part of values.h [142], in particular some of the definitions shown in Listing 6.6.

The macros Tag_val (lines 4/7), Tag_hp (lines 5/8) and Byte_u (line 16) all return an unsigned char * as value. But Bp_hp (line 1), Bp_val (line 12) and Byte (line 15) return a char *. Although not explicitly stated, the definition of Byte as char * and Byte_u as unsigned char * suggests that the macros are intended to return different types. This assumption is supported by the C ISO/IEC standard, which states that "The three types char, signed char, and unsigned char are collectively called the character types. The implementation shall define char to have the same range, representation, and behavior as either signed char or unsigned char" [93, §6.2.5.15]. And according to the GCC compiler documentation, the default representation of char is determined by the ABI, i.e. platform specific [83, §4.4, item 5]. For the AMD64 ABI, for example, both char and signed char are represented as signed byte [97, p. 16, fig. 3.1]. Thus, on a standard machine, the type char * should have the same range, representation and behaviour as signed char *.

However, Linux kernel modules on an average Ubuntu 22.04 x86_64 machine are compiled with the -funsigned-char option, which contradicts these assumptions. For example, in this case `Byte(x, i)` and `Byte_u(x, i)` would have the same type, which may not be the expected result. Thus, when compiling for the Linux Kernel, the expected representation of `char *` should be explicitly specified. Assuming that `char *` shall have the semantics of `signed char *`, the types have been adapted for kernel space compilation. As these macros appear to be unused, it would also be possible to remove them. However, they may be useful in the future.

```c
#define Bp_hp(hp) ((char *) Val_hp (hp))
...
#ifdef ARCH_BIG_ENDIAN
#define Tag_val(val) (((unsigned char *) (val)) [-1])
#define Tag_hp(hp) (((unsigned char *) (hp)) [sizeof(value)-1])
#else
#define Tag_val(val) (((unsigned char *) (val)) [-sizeof(value)])
#define Tag_hp(hp) (((unsigned char *) (hp)) [0])
#endif
...
/* Pointer to the first byte */
#define Bp_val(v) ((char *) (v))
#define Val_bp(p) ((value) (p))
/* Bytes are numbered from 0. */
#define Byte(x, i) (((char *) (x)) [i])
#define Byte_u(x, i) (((unsigned char *) (x)) [i])
```

Listing 6.6: Part of the CertiCoq `values.h`.

The type transformations, e.g. from `nat` or `Z` to `unsigned int` or `int` and vice versa can be implemented as primitive operations. The VeriFFI project [144], for instance, aims to improve this communication and is an ongoing effort to provide a verified foreign function interface for Clight code. This project also contains many examples of type transformations, e.g. between `uint63` and `Z` [143, uint63z] or `uint63` and `nat` [143, uint63nat]. The transformation functions here follow the naming scheme `uint63_from_t` and `uint63_to_t` where `t` is the source and target Coq type, respectively. On the C side, however, `uint63` is a rather unusual type. The usual type would be `uint64_t` (or `int64_t`), for example.

But here again, VeriFFI includes functions to encode and decode between `int63` and `int64_t` [144, c/int63.c]. So it is quite easy to define the transformation between `int64_t` and `Z`, using the examples from VeriFFI.

However, there are two aspects to consider. The first is that the C types `uint64_t` and `int64_t`. This is where `certicoq_stdint.h`, described in Subsection 6.2.1, can be used instead of the includes of `stdint.h` to make the transformations work in both user and kernel space.

A problem arises when using the `uint63_to_z` [143, uint63z/prims.c] transformation function, as shown in Listing 6.7. Here, the expression "`(1 << i)`"(line 4) is interpreted as shifting a signed integer by the unsigned integer value `i`, according to the ISO/IEC C standard [93, §6.4.4.1]. But if the size of the type `value` is 8 bytes, i.e. 64 bits, which is the default case, the expression
`unsigned int i = sizeof(value) * 8 - 1` (line 3)
evaluates to `unsigned int i = 63`. Thus, in the first iteration of the loop, the integer value `1` (which is 32 bits) is shifted left by 63 bits which is a shift overflow.

```
1 value uint63_to_Z(struct thread_info *tinfo, value t) {
2   ...
3   for (unsigned int i = sizeof(value) * 8 - 1; i > 0; i--) {
4     _Bool bit = ((size_t)t & (1 << i)) >> i;
5     ...
6   }
7   return alloc_make_Coq_Numbers_BinNums_Z_Zpos(tinfo, temp);
8 }
```

Listing 6.7: Extract from the function `uint63_to_Z`.

However, no error is provoked when compiling the above example file from VeriFFI using GCC for user space. But when adding the `-fsanitize=shift` compiler flag which is used by default for compiling kernel modules a runtime error (`shift-out-of-bounds`) is generated. The solution here is to add the suffix "ULL" to the integer constant `1`.

With the changes described in this chapter, any synthesised code that does not explicitly use libc functionality or floating-point operations should in principle be a compatible component of a Linux Kernel module. All changes to the CertiCoq files have been applied to a fork [128] of the original CertiCoq repository. Changes to VeriFFI have been applied to a fork [129] of the original VeriFFI repository.

Chapter 7

Discussion

Chapter 5 described how to extend the Coq extraction plugin to support the synthesis of foreign function calls. These changes have the potential to increase the type-safety of a software that contains both synthesised OCaml code and plain OCaml and C code. Consider Figure 7.1, which shows a comparison of the original extraction scheme and the new one.

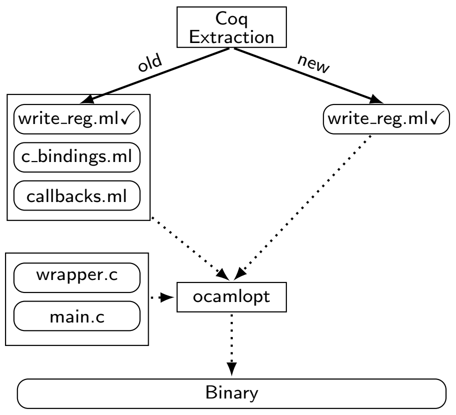

Figure 7.1: Comparison of the old and new extraction scheme.

Here, solid arrows indicate partially trusted processing, while dotted paths indicate untrusted. In the old scheme, both the bindings and the callbacks have to be implemented in additional files, which may only be one file. This can lead to type mismatches between the externals definitions in the `bindings.ml` and `write_reg.ml` if the Coq formalisation changes. In the new scheme, the extraction plugin ensures that the types of the extracted terms match. In both cases, type-safety between the OCaml externals and the C wrapper function cannot be guaranteed. This also applies to OCaml closure definitions and C closure calls.

As pointed out in Section 5.3, there is still potential to improve type-safety between OCaml externals and C wrapper functions by unboxing extraction. Supporting non-allocating extraction for externals would improve the performance of the generated software. In principle, the changes made to OCaml extraction can also be applied to Haskell and Scheme extraction, and potentially to verified OCaml extraction for Coq. But this was beyond the scope of this thesis. The development of the extensions was supported by reviews by Gaëtan Gilbert, Jim Fehrle and Guillaume Melquiond from the Coq team.

The changes described in Chapter 6 make it possible to compile code synthesised by CertiCoq as components of a Linux kernel module. As will be shown later, it will even be possible to use this as a basis for compiling both the synthesised and the glue code using CompCert. However, there are aspects that need to be considered. In Section 6.1, the changes to the garbage collection implementation were described using a schematic approach, which justifies the correctness of the transformations. What still needs to be done, but is beyond the scope of this thesis, is a formal proof of the correctness of the transformations. However, since the transformations lead to a more functional definition of the garbage collection, the proof can be expected to be a moderate effort.

One even more important aspect is that the `garbage_collect` function is called by the synthesised code. In user space, both the original and the lazily failing garbage collection will eventually exit on failure. But this is not true for the kernel space variant, since the kernel variant cannot exit without stopping the whole system, as described in Section 2.1. Also, the "non-local jump" seems to be unusable. And this is a problem that cannot be solved without changes to the CertiCoq code generation. Although the new `garbage_collect` function returns `false` on error, no check on the return value is synthesised by CertiCoq. Conceptually, a possible approach could be that when synthesising a function $f : A \to B$ (with $b \in B$) containing the garbage collection call, the function should be synthesised to return the type *option B*. Then the result of f should be *None* if the garbage collection failed and *Some b* otherwise. Furthermore, any function calling f would then also have to check whether f returned *None* and propagate that error value. In principle, the outermost function called by plain C code could then also return *None*, which would make it possible to detect a garbage collection failure. But this approach would inevitably introduce additional runtime complexity. And since this is only necessary for kernel space applications, this synthesis should be explicitly selectable, e.g. with a new CertiCoq option.

But even without actually handling a garbage collection error, a proof of concept can be given. The following part will present a partially synthesised Linux device driver. This is done by first deriving a functional model from the original code, and then showing how the synthesised code can actually be compiled for kernel space.

Part III

A Partially Synthesised Device Driver

Chapter 8

From Code to Formalisation

As mentioned in Section 2.2, the PC Speaker driver [116] is both a platform device driver and manages an input device. So the callback function `event` is set to the function `pcspkr_event`. The event function takes as input a pointer to the input device, an unsigned integer `type` and `code`, and a signed integer `value`, as shown in Listing 8.1, copied[1] from the Linux Kernel GitHub repository [116].

As mentioned earlier, there is usually no formal specification of how a driver should behave. And also in this case, the original driver does not reference any specification. So both a functional model and its formal properties have to be derived from the original source code. This was done in the course of the PhD project formalisation [127], and the following description briefly discusses its key points. The approach was to first manually derive a functional model and then state the logical properties and prove that the functional model actually satisfies them.

In the following, references to lines always refer to the original C code. Apart from the fact that the device pointer is unused in this case, the `type` can be used to determine whether the driver should react to the event. The `code` then identifies the driver mode, while the `value` represents the frequency value.

```
1  static int pcspkr_event(struct input_dev *dev, unsigned int type,
2                          unsigned int code, int value)
3  {
4    unsigned int count = 0;
5    unsigned long flags;
6
7    if (type != EV_SND) return -EINVAL;
8
9    switch (code) {
```

[1]With slight modifications to the layout.

```
    case SND_BELL: if (value) value = 1000; break;
    case SND_TONE: break;
    default:       return -EINVAL;
    }

    if (value > 20 && value < 32767)
      count = PIT_TICK_RATE / value;

    raw_spin_lock_irqsave(&i8253_lock, flags);

    if (count) {
      /* set command for counter 2, 2 byte write */
      outb_p(0xB6, 0x43);
      /* select desired HZ */
      outb_p(count & 0xff, 0x42);
      outb((count >> 8) & 0xff, 0x42);
      /* enable counter 2 */
      outb_p(inb_p(0x61) | 3, 0x61);
    } else {
      /* disable counter 2 */
      outb(inb_p(0x61) & 0xFC, 0x61);
    }

    raw_spin_unlock_irqrestore(&i8253_lock, flags);

    return 0;
  }
```

Listing 8.1: The original code of the PC Speaker event handler.

Since `type` and `code` are unsigned integers, and thus can be formalised as values of the Coq type `nat`, the frequency `value` is signed and can be represented in the Coq type `Z`. However, for performance reasons, unsigned integers should rather be represented by the Coq type `N`. Also, the function shall return a signed integer, which is also represented as type `Z`.

The first property that can be derived is that the event handler shall return the value `-EINVAL` (i.e. -22 [109]) if and only if the `type` is not `EV_SND` (i.e. `0x12`/18 [112]) or if the `code` is neither `SND_BELL` (i.e. `0x01`/1) nor `SND_TONE` (i.e. `0x02`/2). Otherwise the handler shall return 0.

And indeed, the event handler first checks whether the event `type` is the expected one, i.e. `EV_SND`, and returns `-EINVAL` if it is not. This can be easily modelled in Coq, as shown in Listing 8.2.

```
Definition pcspkr_evt (type code : nat) (value : Z) : Z :=
  if (type =? 0x12)
  then go_on code value
  else −22.
```

Listing 8.2: Partial formalisation of `pcspkr_event`.

Here the function `go_on` is a dummy function that will be refined later, and the value `EV_SND` is unfolded according to its definition. It then checks whether `code` is one of the two supported values and returns `-EINVAL` if it is not. In lines 10 and 20 it is checked whether some integer values, here `value` and `count`, evaluate to the boolean value true. According to the ISO/IEC specification, an integral value is evaluated as true in C if and only if it is non-null [93, §6.8.4.1, Sentence 2] in an `if` statement. However, since `value` and `count` have different types, i.e. signed (`Z`) and unsigned (`nat`), two definitions are needed to model this, as shown in Listing 8.3. In both cases, the function evaluates to true if and only if the given value does not match with `O` or `Z0`. The specification for `z_true_in_c` to be proved is also given in the listing.

```
Definition nat_true_in_c ( a : nat ) : bool :=
  match a with
   | O  ⇒ false
   | _  ⇒ true
  end.

Definition z_true_in_c ( a : Z ) : bool :=
  match a with
   | Z0 ⇒ false
   | _  ⇒ true
  end.

Lemma z_true_in_c_spec : ∀ ( val : Z ), val ≠ 0 ↔ z_true_in_c val = true.
```

Listing 8.3: Definitions of "truth" according to the ISO/IEC C specification.

The overall semantics of the switch statement (lines 9–13) is twofold: to return an error value if necessary, and to manipulate the `value` input parameter if `code` is `SND_BELL` (`0x01`) and `value` is not zero. This means that in the `SND_BELL` mode `value` will always be either 1000 or 0. In `SND_TONE` (`0x02`) mode, `value` will remain unchanged. Thus, the switch statement can be modelled by the function it computes, which is computing the `value` from the `code`, as shown in Listing 8.4.

```
Definition filter_val (val : Z) : Z :=
  if (z_true_in_c val) then 1000%Z else val.

Definition snd_of_nat (code : nat) : option snd :=
  if (code =? 0x01) then Some Bell else
  if (code =? 0x02) then Some Tone
  else None.

Definition val_of_code (code : nat) (val : Z) : option Z :=
  match (snd_of_nat code) with
  | Some Bell ⇒ Some (filter_val val) (*Line 10*)
  | Some Tone ⇒ Some (val)            (*Line 11*)
  | None      ⇒ None                  (*Line 12*)
end.
```

Listing 8.4: Computing the value from the code.

The `snd_of_nat` function maps a natural number to an option type, where the result is `Some a` if `code` is valid and `None` otherwise. The function `val_of_code` then propagates `None` for an undefined code and returns the `value` wrapped in `Some` in the other cases. While in the `Tone` case it returns the original `value`, in the `Bell` case it returns 1000 if and only if the value was not zero. Now `pcspkr_evt` can be refined by replacing `go_on` as shown in Listing 8.5.

```
Definition pcspkr_evt (type code: nat) (val : Z) : Z :=
  if (type =? 0x12) then
    match (val_of_code code val) with
    | Some a ⇒ go_on2 a
    | None   ⇒ −22
    end
  else −22.
```

Listing 8.5: First refinement of `pcspkr_evt`.

This refinement uses pattern matching on the result of `val_of_code` and returns `-EINVAL` if the code was invalid and compute the `count` (line 16) otherwise. On this basis, the specification of when the functional model will return `-EINVAL` can be given as shown in Listing 8.6.

```
Lemma pcspkr_evt_einval_spec : ∀ (type code : nat) (val : Z ),
  is_ev_snd type = false ∨ val_of_code code val = None
  ↔ pcspkr_evt type code val = einval.
```

Listing 8.6: First property of the first refinement of `pcspkr_evt`.

The behaviour after the switch statement can now be formalised. Initially, the variable `count` is set to 0 (in line 4) and the variable is only changed if `value` is in the interval $[21, 32766]$ by computing $PIT\_TICK\_RATE/value$ (line 16). Setting the count can be modelled as shown in Listing 8.7.

```
Definition is_valid_value (val : Z) : bool :=
  (20 <? val) && (val <? 32767).

Definition set_count (val : Z) (count : nat) : nat :=
  if (is_valid_value val) then PIT_TICK_RATE / (Z.to_nat val)
  else count.
```

Listing 8.7: Computing the count from the value.

The test of whether `value` is in the interval is modelled by a less relation, and the `count` is computed in the same way as in the original code. The transformation of `value` to `nat` is indeed safe here, since the division is only computed if and only if `value` is in the frequency interval, which includes being non-negative. However, `set_count` must take 0 as `count` parameter, as in the second refinement of `pcspkr_evt` shown in Listing 8.8.

Here, `set_count` is computed and passed to the `sound_switch` function, which is a dummy that will be refined in the following. This function represents the conditional in line 20 of the original code and then either activates (lines 21–27) or deactivates (lines 29–30) the PC Speaker.

```
Definition pcspkr_evt (type code: nat) (val : Z) : Z :=
  if (type =? 0x12) then
    match (val_of_code code val) with
    | Some a ⇒ sound_switch (set_count a 0)
    | None   ⇒ −22
    end
  else −22.
```

Listing 8.8: Second refinement of `pcspkr_evt`.

It is important to note that the conditional on `count` is wrapped in a lock guard, i.e. a lock is acquired before entering the conditional and released after leaving it. However, as the locking is rather a side effect (the calls to `raw_spin_lock_irqsave` and `raw_spin_unlock_irqrestore` are actually macros that return nothing [117, lines 241ff and 279ff]), it is not possible to formalise them purely functionally.

For formalisation, locking and unlocking has been defined as an axiomatic function `Axiom linux_lock : Z → Z`. This function takes an input specifying whether a lock is to be acquired (value 1) or released (value 0). The C-side implementation of the function will acquire or release the lock accordingly, always returning the value 0. While this is not an optimal solution, it was the simplest for preliminary testing. A better solution would be to use, for example, the IO monads from `coq-simple-io` [107]. But this was beyond the scope of this thesis. Also, the functions `outb` and `outb_p`, which write an 8-bit word into a given register, do not return anything [115, lines 581, 624 and 661]. The functions `outb`, `outb_p`, `inb`, `inb_p` and also the locking are therefore considered as primitive functions for the purpose of this document and can be defined as axioms. And it is important to note that the functions must be executed in this order. However, it is possible to prove that the functions `outb` and `outb_p` are correctly used in the original code by looking again at the sound switching conditional in Listing 8.9.

```
 1 if (count) {
 2   /* set command for counter 2, 2 byte write */
 3   outb_p(0xB6, 0x43); // 182
 4   /* select desired HZ */
 5   outb_p(count & 0xff, 0x42);
 6   outb((count >> 8) & 0xff, 0x42);
 7   /* enable counter 2 */
 8   outb_p(inb_p(0x61) | 3, 0x61);
 9 } else {
10   /* disable counter 2 */
11   outb(inb_p(0x61) & 0xFC, 0x61);
12 }
```

Listing 8.9: The sound switch conditional.

As already noted, the `outb` functions expect 8-bit unsigned integers, but the count is an unsigned integer and therefore 64 bits wide on an x86_64 architecture. And for unsigned 8-bit integers, the maximum value is 255.

The first call to `outb_p` writes the value `0xB6`, which is 182 in decimal notation and therefore in range. In lines 5 and 6, a bitwise `and` with `0xff` (255) is applied to `count` and `count >> 8`, which sets all but the last 8 bits to 0. Obviously, the semantics of these lines is to write the last byte first and then the previous byte into register `0x42`, as shifting by 8 removes the last byte. In line 8, the PC Speaker register (`0x61`) is read (which yields an unsigned 8-bit value) and this value is written back after applying a bitwise `or` with 3 (binary 11). This sets the last 2 of the 8 bits and the value written has an 8-bit word size. For disabling the sound, the PC Speaker register is read and a bitwise `and` with `0xFC`, which is binary 11111100, is written back, i.e. the last two bits are unset. All these properties were proven during the PhD project.

The behaviour of the sound switch can then be formalised in Coq, as shown in Listing 8.10.

```
Definition sound_switch (count : nat) :=
  if nat_true_in_c count
  then enable_sound (last_byte count) (prev_byte count)
  else disable_sound 0x61

Definition pcspkr_evt (type code: nat) (val : Z) : Z :=
  if (type =? 0x12) then
    match (val_of_code code val) with
    | Some a ⇒ sound_switch (set_count a 0)
    | None   ⇒ invalid_val
    end
  else invalid_val.
```

Listing 8.10: Computing the value from the code.

The function `sound_switch` calls `enable_sound` and `disable_sound` according to the original definition in the C code. But here `enable_sound` can take the last and the previous byte of the `count` as parameters, while `disable_sound` takes the PC Speaker register as input. Both `last_byte` and `prev_byte` can be easily formalised in Coq. The former is in fact nothing more than `Nat.shiftr code 8`, while the latter is `last_byte (Nat.land 0xff count)`. However, `enable_sound` and `disable_sound` can be defined as axiomatic functions that return a fixed value of 0. Returning a fixed value is safe here, since the calls to `outb` and `outb_p` and the locking do not return anything. Additionally, the C implementations used as primitives have to do the locking. This way, `pcspkr_evt` will return −22 if and only if either `type` or `code` are invalid, and 0 otherwise.

Alternatively, and as done in the PhD project, `enable_sound` can be formalised as shown in Listing 8.11.

```
Definition enable_sound (last_byte prev_byte : nat) : Z :=
  if ((out8_p snd_command Cmd) =? 0) then
    if ((out8_p last_byte Data2) =? 0) then
      if ((out8 prev_byte Data2) =? 0) then
        (out8_p (set_last_2_bits (in8_p Pc_spk)) Pc_spk)
      else 1
    else 1
  else 1.
```

Listing 8.11: Conditional based formalisation of `enable_sound`.

Here the commands are executed in a cascade of conditionals, always expecting to have a return value of 0 on successful execution of the primitives and 1 on error. Since Coq itself is not aware of the fact that the primitives always return 0, this formalisation enforces sequential execution (even when extracting to OCaml).

To ensure that a lock is acquired before writing to registers and released afterwards, a wrapper function can be defined to do the locking and execute `enable_sound`, as shown in Listing 8.12.

```
Definition enable_sound_locking (lb pb : nat) : Z :=
  if linux_lock 1 =? 0 (* Acquire Lock*)
  then
    let result := enable_sound lb pb in
    if (result =? 0) (* enable_sound success, return unlock result *)
    then linux_lock 0
    else (* enable_sound failed with 1, try unlock*)
      if linux_lock 0 =? 0
      then result
      else 1 (* Unlocking failed *)
  else 1. (* Locking failed*)
```

Listing 8.12: Conditional based formalisation of locking `enable_sound`.

Again, the locking, execution of `enable_sound` and unlocking is done in a cascade of conditionals, enforcing sequential execution. It is possible to prove that the `enable_sound_locking` function always returns 0. But this requires two assumptions: That `linux_lock` always returns 0, and that `outb` and `outb_p` always return either 0 or 1. Both assumptions are valid, since according to the Linux kernel documentation, the locking always succeeds and the register write operations return nothing. And in both cases, the calls to the concrete Linux implementations can be wrapped in a primitive that always returns 0. Nevertheless, a goal of future work is to formalise the sound switch without these assumptions, e.g. by using IO monads.

On this basis, the following chapter shows how to construct a working Linux kernel space device driver.

Chapter 9

Constructing a Linux Device Driver

Given the formalisation, CertiCoq can be used to synthesise the Clight code and the necessary glue code. However, more is required to construct a Linux device driver. The description here briefly follows the author's prototypical implementation [131], which has been successfully tested both in an emulated environment and on native hardware. For instance, parts of the original implementation [116] are needed. Apart from the `pcspkr_event` function formalised in the last chapter, the implementation contains necessary includes of Linux kernel headers that can be reused without modification. Additionally, the implementation contains static functions that define the callbacks (probe, remove, suspend, shutdown) and static structures that define the power management options (`pcspkr_pm_ops`) and the platform driver (`pcspkr_platform_driver`). While these functions and structures can remain unchanged, the `pcspkr_event` function can be eroded, i.e. redefined to merely call an external function (e.g. defined in `clight_iface.c`) that calls the Clight code. A restructured snippet of this function is shown in Listing 9.1. The code first initialises the `thread_info` using the `make_tinfo` function from the CertiCoq garbage collection adapted in Chapter 6.1 and returns `-EINVAL` if the `thread_info` could not be created. It then uses the VeriFFI type transformations adapted in Subsection 6.2.3 to transform the parameters to `nat` and `Z` respectively.

```
struct thread_info *ti = make_tinfo();
if (ti == 0) return -22;

value type_coq = int64_to_nat(ti, type);
value code_coq = int64_to_nat(ti, code);
value val_coq  = int64_to_Z(ti, val);

value tmp = body(ti);
tmp = call(ti, tmp, type_coq);
tmp = call(ti, tmp, code_coq);
tmp = call(ti, tmp, val_coq);

int result = int64_from_Z(tmp);
free_tinfo(ti);
return result;
```

Listing 9.1: The code initialising and calling the synthesised code.

The `thread_info` is then used to instantiate the functionality of the synthesised function using the function `body` from the synthesised Clight file. To start the synthesised function, it is applied to the Coq type representations using the function `call` from the synthesised glue code. When the last value, i.e. `val_coq`, is applied, the synthesised function is automatically evaluated. Finally, the result of the function, which must be either 0 or -22 according to the formalisation, is transformed into a signed integer and returned after freeing the heap and `thread_info` using the function introduced in 6.1.5.

Thus, the driver needs at least the eroded original C file, the Clight interface file and the garbage collection and runtime from CertiCoq. Additionally, all primitive functions have to be defined in C (either plain or synthesised), including register I/O functions, locking and type transformations. The latter can be done by reusing functionality from VeriFFI.

Given that the main file, i.e. the eroded original implementation, has been renamed to `certpcspkr_mod.c`, a `Makefile` must be created for this module according to the descriptions in Section 2.2. As described, the `Makefile` must contain at least the definition of `obj-m`, i.e. the name of the output object that will form the kernel object. However, in this use case, some additional definitions need to be provided, as shown in Listing 9.2. That is, all the necessary object files are added sequentially to an `ADD_OBJS` list (lines 3–8). These are the objects from the synthesised code, the CertiCoq and VeriFFI files, the code called by the synthesised code (line 6), the implemented low-level functionality and the interface between the module and the synthesised code. Then `certpcspkr-objs` is defined and the prefix `certpcspkr` tells the compilation toolchain that the named objects together form the `certpcspkr.o` object.

Also, additional compiler flags are required. Noteworthy are the definitions `CLIGHT_KERNEL_CODE=1` and `CERTICOQ_TRACE=1`. While the former ensures that the `printtree` function of the garbage collection is excluded, the latter is used to enable tracing in the CertiCoq primitives file as described in Subsections 6.1.5 and 6.2.1. Although the synthesised code can be expected to be correct, it contains switch statements that are in fact empty[1]. And while all synthesised functions are supposed to return a `value` type result, these expressions return nothing. The GCC, for example, detects this. While this is normally only signalled by a warning, Linux kernel modules are compiled by default to treat this warning as an error. And since this missing return statement can be considered irrelevant, as the code synthesised by CertiCoq can be expected to be correct, the flag `-Wno-error=return-type` has to be set to allow successful compilation.

```
obj-m += certpcspkr.o

ADD_OBJS  = pcspkr_evt_linux.o glue.o
ADD_OBJS += gc_stack.o kmod_alloc.o prim_int63.o
ADD_OBJS += int6x_nat.o int6x_z.o int64.o
ADD_OBJS += nat_ffi.o regio_ffi.o locking_ffi.o z_ffi.o
ADD_OBJS += regio.o locking.o pit.o
ADD_OBJS += clight_iface.o

certpcspkr-objs := certpcspkr_mod.o $(ADD_OBJS)

EXTRA_CFLAGS  = -O2 -Wno-error=return-type -I$(PWD)
EXTRA_CFLAGS += -D CLIGHT_KERNEL_CODE=1 -D CERTICOQ_TRACE=1

all:
	make -C /lib/modules/$(shell uname -r)/build M="$(PWD)"\
	CONFIG_FUNCTION_TRACER= modules
```

Listing 9.2: Relevant Makefile contents for GCC compilation.

With the described definitions in the `Makefile`, it is possible to compile the synthesised code into a working Linux kernel module. Moreover, the synthesised part can be expected to be correct up to source level, which is definitely an improvement. Thus, the research question of whether it is possible to construct at least partially certified Linux kernel modules using CertiCoq is answered in the affirmative.

However, this result is rather limited, since much more is possible for user space code, as described in Subsection 3.2.2. That is, if CompCert is used to compile the synthesised Clight code, the correctness of the generated assembly code can be expected. Thus, an obvious question is whether it would be possible to compile Linux kernel modules at least partially using CompCert, i.e. at least for the synthesised Clight code. As discussed in Subsection 6.2.3, the `-funsigned-char` and `-fsanitize=shift` options are used by default by the Linux Kernel module

[1]Examples can be easily found in the file `synthesised/pcspkr_evt_linux.c` [131].

compilation toolchain. However, according to the CompCert manual [96], these options are not supported. There are also other options used by the toolchain, such as `-MMD` and `-fno-PIE`, which appear to be unsupported by CompCert. So it will not be possible to use CompCert to compile the whole module. But this is where the functionality to provide shipped objects described in Section 2.2 comes into play. If it were possible to compile the synthesised code and the glue code with CompCert, and give the resulting object files the `o_shipped` extension, it could be possible to link those files to the module. But as pointed out, a `.cmd` file must accompany all object files after compilation, and must be created separately for shipped objects. Thus, the `Makefile` needs to be extended as shown in Listing 9.3.

```
%.o_shipped : %.c
  ccomp -c -o $@ -D CLIGHT_KERNEL_CODE=1 -D CERTICOQ_TRACE=1\
        -I$(PWD) $<

%.o_shipped.cmd : %.o_shipped
  touch .$(basename $^).o.cmd
  touch $@

CCOMP_OBJS      := pcspkr_evt_linux.o_shipped glue.o_shipped
CCOMP_OBJS_CMD  := pcspkr_evt_linux.o_shipped.cmd glue.o_shipped.cmd

with_compcert : $(CCOMP_OBJS) $(CCOMP_OBJS_CMD)
  make -C /lib/modules/$(shell uname -r)/build M="$(PWD)"\
        CONFIG_FUNCTION_TRACER= modules
```

Listing 9.3: Relevant Makefile contents for CompCert compilation.

Here, the `with_compcert` target expects the existence of the shipped objects and the accompanying `cmd` files. The make system finds a rule for creating shipped object files from C files, which invokes CompCert to produce exactly those files. The make system also finds a rule that creates a `cmd` file for each shipped object by touching[2] the `.f.o.cmd` file. The rule also touches `f.o_shipped.cmd` to indicate that the `cmd` file has already been created. And indeed, with these modifications, the toolchain can use CompCert to compile the synthesised and glue code, while the rest is compiled with GCC. The result is a fully functional Linux Kernel module - with the advantage that the assembly code generated by CompCert can be expected to be correct.

While it is now evident that synthesised and glue code can be compiled with CompCert for kernel space, it has not yet been discussed whether this also holds for the primitives reused from CertiCoq or VeriFFI. But this can be answered

[2]The tool `touch` creates a file, if it does not exist and updates the last access time otherwise.

quite easily. As CompCert is a certified compiler for C (and Clight) code, the compilation of primitives will be possible if they are implemented in this fragment of C and do not use unsupported preprocessor functionality. Indeed, compilation of the file `c/int6x_z.c` from the VeriFFI fork [129] presented in Subsection 6.2.3 is possible without any problems. However, the attempt of compiling the main C file, i.e. `certpcspkr_mod.c` with CompCert was not successful and failed with multiple preprocessor errors.

Another important aspect when using synthesised components in kernel space is the limited stack size, as described in Section 2.1. Synthesising non-tail recursive functions can easily exceed the stack size and freeze the whole system. An example of such a function is the transformation of values of type `Z` to `nat`, which can be achieved with the function `Z.to_nat`[3]. This function expands to `Pos.to_nat p` in the positive case, which in turn expands to `Pos.iter_op Nat.add x 1`. And this is where deep recursion is used, i.e. `iter_op` is defined by a recursive function `iter`, which uses `Nat.add`. And `Nat.add` itself is also defined recursively. Concluding, synthesising such functions should be avoided, e.g. by using primitive functions instead. In the following, some considerations about performance are given.

[3]From the module `Coq.ZArith.BinInt`

Chapter 10

Performance Evaluation

As mentioned in Subsection 3.2.2, the runtime performance of code synthesised by CertiCoq is not as good as natively compiled and synthesised OCaml code. In fact, the performance of a synthesised driver depends on two aspects: The formalisation and the optimisations performed by CertiCoq. So it is quite obvious that the partially synthesised PC Speaker driver [131] will not be as performant as the original one. Nevertheless, some preliminary performance evaluations have been made. Here, it is important to note that the synthesised driver has to use a primitive operation to compute PIT_TICK_RATE / $value$ in order not to provoke a stack overflow. This makes the results better than they would be if the computation was done in synthesised code.

The benchmarks were run in a virtual machine in QEMU with Ubuntu 22.04 operating system and the Linux Kernel 6.5.0-35-generic. Both the original and the synthesised drivers were compiled with optimisation level 2 (`-O2`) and GCC 12.3.0. To measure the runtimes, the source code was extended with measurement functionality using `ktime` [68]. Specifically, the duration for enabling the sound was measured in nanoseconds for both the `Bell` (`SND_BELL`) and `Tone` (`SND_TONE`) cases using the `beep`[1] [136] tool. For both modes, 1000 executions were measured and the runtimes were stored in an array at runtime, as writing to files in kernel space is discouraged, as mentioned. After 1000 executions for each mode, the minimum, maximum and median runtimes were printed to the Kernel log, along with the sum of all runtimes. The sum was used because floating-point operations are discouraged in kernel space, and computing the average of all executions would mean truncating the result to an integer, which significantly reduces precision. After each execution, a delay of 100 milliseconds was introduced.

The following Table 10.1 shows the minimum, average, median and maximum

[1]For testing the `Bell` case, a patched version of the tool was used that emits `SND_BELL` instead of `SND_TONE`.

runtimes for enabling the sound for the `Bell` and `Tone` modes of both the original and the synthesised PC Speaker driver and, all times given in microseconds.

Driver	**Mode**	**min** (μs)	**avg** (μs)	**med** (μs)	**max** (μs)
C	Bell	30.79	307.39	72.82	28829.74
Synth	Bell	74.99	197.60	182.81	13073.84
C	Tone	30.18	100.81	75.13	16578.81
Synth	Tone	87.54	224.25	225.09	1322.05

Table 10.1: Benchmark results for the original and synthesised driver.

As expected, the synthesised driver does not compete well with the original. The table shows that the synthesised driver has a 144 to 150 % higher best and median runtimes in the `Bell` mode. In the `Tone` mode, both the best and median runtimes are about 190 % higher than those of the original driver. There is also a fairly large difference between the average and median runtimes and between the best and worst case runtimes, and there are several reasons for this. One reason is the locking: While the Linux kernel guarantees that the lock will be acquired, there is no guarantee when the lock will be acquired, i.e. acquiring a lock can take some time. Also, in some cases the scheduler may stop the driver or reassign it to another CPU, causing latency. And finally, the garbage collection probably has some impact on runtime performance. All these effects are not easy to filter out when using `ktime`. This would require detailed benchmarks using `ftrace` [64]. However, they would not provide much more insight than what is discussed below.

These results already show some aspects that can and must be improved. For instance, the use of the Coq `nat` datatype is obviously inappropriate and Coq has a much more efficient datatype for unsigned values, which is `N`. Also, it was pointed out in Chapter 8 that in the `Bell` mode the `value` is always set to either 1000 or 0. And if the `value` is 1000, the sound is enabled, whereas if the value is 0, the sound is disabled. This means that there is no need to check whether the value is in a valid frequency interval. And this also means that the computation *count = PIT_TICK_RATE / value* is actually a division of constants in the `Bell` mode. From this, it follows that getting the last and the previous byte of the `count` is an operation on constants. Concluding, at least for the `Bell` case, the functional model can be significantly optimised. Apart from that, the performance can be improved by improving the code generation of CertiCoq. In the following, the results of this thesis are discussed.

Chapter 11

Conclusion

In Subsection 3.2.1, it was shown that it is not possible to use OCaml code synthesised by the Coq extraction plugin to produce working Linux kernel space drivers, while it is obviously possible to produce working user space drivers as long as a libc can be used. It was discussed that this limitation also applies to other languages supported by the Coq extraction plugin.

It was also shown that for the user space approach, type-safety can be improved by synthesising foreign function definitions from Coq formalisations. However, since the Coq extraction plugin is not verified, the gain would be rather limited. Here, the verified OCaml synthesis would be a more appropriate pipeline, and the changes made to the Coq extraction plugin could be applied to this pipeline in the future.

The modifications to the CertiCoq runtime and garbage collection presented in Chapter 6 were introduced to create a kernel space compatible runtime by applying program transformation to the garbage collection. Necessary modifications to the glue code, synthesised code and primitives provided by VeriFFI were also presented.

On this basis, a proof of concept for the construction of Linux kernel space device drivers using code synthesised by CertiCoq was given in this part. First, the original event handling functionality of the Linux PC Speaker driver was gradually formalised. Then it was shown how to combine the synthesised code and the glue code with plain C functionality and compile both with GCC into a kernel object using GCC. This showed that it is indeed possible to construct at least partially certified Linux kernel space device drivers from Coq formalisations.

It was then shown that it is also possible, in principle, to extend the partial certification to the assembly code being produced by using shipped objects compiled by CompCert from the synthesised and the glue code. The implication is that the correctness guarantee which holds for user space software can also be applied to kernel space software. However, it was also shown that there are limitations, i.e. the stack size in kernel space and floating-point operations.

This proof of concept is clearly a starting point for future work. For example, as many primitives as possible should be made kernel space compatible, and some functionality has already been ported [128]. This would increase the ratio of code that can be compiled using CompCert. Moreover, it should be possible to adapt the changes on the runtime and garbage collection presented in Chapter 6 to other operating systems like NetBSD[1] or FreeBSD[2], with limited effort. And most importantly, the changes to CertiCoq's garbage collection have opened a weak point as the result of the garbage collection call is not evaluated by the synthesised code. While it has been shown to be necessary to use a lazily failing garbage collection in kernel space, this weak point has yet to be addressed, and a possible solution was given in Chapter 7.

Also, as pointed out in Subsection 3.2.2, the runtime performance of code synthesised by CertiCoq is not as good as that of natively compiled and synthesised OCaml code. But especially for kernel space software, performance is important. And it was discussed in Chapter 10 that the initial proof of concept has a weak performance and that both improvements in the functional model and in the code generation of CertiCoq will have an effect on the performance. Regarding code generation, it is already possible to exclude the garbage collection for functions that do not allocate by omitting `with tinfo` in the **`CertiCoq Register`** [99]. This is similar to the `@@noalloc` annotation in OCaml discussed in Section 5.3. But all values of type `Z` or `nat`, for example, are provided as boxed values when primitive functions are called. That is, a value of type `nat` is represented by a pointer to the heap memory, which contains a recursive data structure representing the value. But transforming this value into `uint63` has linear time complexity. So it is an open question whether this overhead can be reduced.

[1]`https://www.netbsd.org`
[2]`https://www.freebsd.org`

Part IV

Used Sources

Chapter 12

Introduction

This part contains a description of the used sources, i.e. literature and artifacts. To make the used sources more strictly separated, the usual bibliography is divided into three chapters. Chapter 12 lists the scientific literature, i.e. books, articles, proceeding contributions, theses and comparable sources. In Chapter 13, reference manuals, documentations and comparable entries are listed which also includes email communication (e.g. relevant emails by Linus Torvalds) and repository issues. Finally, Chapter 14 lists software artifacts which may have a DOI, if they are available via services as Zenodo[1]. This appendix also includes direct URLs to repositories on GitHub, for example. All entries in Chapter 13 and 14 have a last access timestamp, if no DOI exists. For documentations for some specific software versions, the version release date is used as publication date, if the respective website does not specify a copyright date explicitly. For repositories, it is defined by the last change made on that specific entry. Thus, although the Linux Kernel v6.7 was published in 2024, a source that belongs to this version may have an earlier publication date, if the change was prior to the publication of the Linux Kernel itself. The LaTeX template used for writing this thesis was provided by Dr. Sebastian Böhne. The Coq listings were typeset using the Coq listings definitions of Assia Mahboubi[2]. This definition was extended to support new commands introduced in this thesis and CertiCoq specific commands.

[1] `https://zenodo.org/`

[2] `http://people.rennes.inria.fr/Assia.Mahboubi/`

Chapter 13

Literature

[1] Hemant Agrawal & Ravi Malhotra (2012): *Device Drivers in User Space: A Case for Network Device Driver.* *International Journal of Information and Education Technology*, pp. 461–463, doi:10.7763/IJIET.2012.V2.179.

[2] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau & Matthew Weaver (2017): *CertiCoq : A verified compiler for Coq.* In: *CoqPL'17: The Third International Workshop on Coq for Programming Languages*. Available at `https://www.cs.princeton.edu/~mzweaver/pdfs/CoqPL17.pdf`. (Last access: Mar. 18 2024).

[3] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen & Bas Spitters (2022): *Extracting functional programs from Coq, in Coq.* *Journal of Functional Programming* 32, p. 11, doi:10.1017/S0956796822000077.

[4] Andrew W. Appel (1991): *Compiling with Continuations.* Cambridge University Press, doi:10.1017/CBO9780511609619.

[5] Andrew W. Appel (2023): *Verifiable C.* Available at `https://github.com/PrincetonUniversity/VST/raw/master/doc/VC.pdf`. (Last access: Mar. 18 2024).

[6] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich & Steve Zdancewic (2017): *Position paper: the science of deep specification.* *Phil. Trans. R. Soc.* 375(2104), doi:10.1098/rsta.2016.0331.

[7] Andrew W. Appel & Trevor Jim (1997): *Shrinking lambda expressions in linear time.* *J. Funct. Program.* 7(5), p. 515–540, doi:10.1017/S0956796897002839.

[8] Christel Baier & Joost-Pieter Katoen (2008): *Principles of model checking.* The MIT Press.

[9] Olivier Savary Bélanger (2019): *Verified Extraction for Coq.* Ph.D. thesis, Princeton University. Available at `https://www.cs.princeton.edu/techreports/2019/011.pdf`. (Last access: Mar. 18 2024).

[10] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* doi:10.1007/978-3-662-07964-5.

[11] Sandrine Blazy & Xavier Leroy (2009): *Mechanized Semantics for the Clight Subset of the C Language.* *Journal of Automated Reasoning* 43(3), pp. 263–288, doi:10.1007/s10817-009-9148-3.

[12] B. Chess & J. West (2007): *Secure Programming with Static Analysis.* Addison-Wesley software security series, Addison-Wesley.

[13] Adam Chlipala (2007): *A certified type-preserving compiler from lambda calculus to assembly language. SIGPLAN Not.* 42(6), p. 54–65, doi:10.1145/1273442.1250742.

[14] Adam Chlipala (2010): *A verified compiler for an impure functional language.* In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, Association for Computing Machinery, New York, NY, USA, p. 93–106, doi:10.1145/1706299.1706312.

[15] T. Coquand & C. Paulin (1990): *Inductively defined types.* In: *Proceedings of the International Conference on Computer Logic*, COLOG-88, Springer-Verlag, Berlin, Heidelberg, p. 50–66, doi:10.1007/3-540-52335-9_47.

[16] Jonathan Corbet, Alessandro Rubini & Greg Kroah-Hartman (2005): *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc.

[17] Stephen Dolan (2016): *Malfunctional programming.* In: *ML Workshop.* Available at `https://stedolan.net/talks/2016/malfunction/malfunction.pdf`. (Last access: Mar. 18 2024).

[18] Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl & Georg Carle (2019): *User Space Network Drivers.* In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 1–12, doi:10.1109/ANCS.2019.8901894.

[19] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood & Adam Chlipala (2021): *Integration verification across software and hardware for a simple embedded system.* In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, Association for Computing Machinery, New York, NY, USA, p. 604–619, doi:10.1145/3453483.3454065.

[20] Yannick Forster, Matthieu Sozeau & Nicolas Tabareau (2023): *Verified Extraction from Coq to OCaml.* Available at `https://inria.hal.science/hal-04329663`. Working paper or preprint.

[21] Nicolas Frinker, Steffen Liebergeld, Dr. Andreas Otto, Mario Frank & Mario Egger (2023): *Eine vertrauenswürdige, sichere Public Cloud - Utopie oder Realität?* In: *Digital sicher in eine nachhaltige Zukunft*, 19. Deutscher IT-Sicherheitskongress, Bundesamt für Sicherheit in der Informationstechnik - BSI, pp. 227–240.

[22] Kai Germaschewski & Sam Ravnborg (2003): *Kernel configuration and building in Linux 2.5.* In: *Proceedings of the Ottawa Linux Symposium.* Available at `https://www.kernel.org/doc/ols/2003/ols2003-pages-185-200.pdf`. (Last access: Mar. 18 2024).

[23] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg & David Costanzo (2016): *CertiKOS: an extensible architecture for building certified concurrent OS kernels.* In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, USENIX Association, USA, p. 653–669. Available at `https://dl.acm.org/doi/10.5555/3026877.3026928`. (Last access: Mar. 18 2024).

[24] Joshua D. Guttman, John D. Ramsdell & Vipin Swarup (1995): *The VLISP verified Scheme system* 8, pp. 33–110. doi:10.1007/BF01128407.

[25] Paul Hamill (2004): *Unit Test Frameworks: Tools for High-Quality Software Development.* O'Reilly Media.

[26] C. A. R. Hoare (1969): *An axiomatic basis for computer programming. Commun. ACM* 12(10), p. 576–580, doi:10.1145/363235.363259.

[27] Lars Hupel & Tobias Nipkow (2018): *A Verified Compiler from Isabelle/HOL to CakeML.* In Amal Ahmed, editor: *European Symposium on Programming (ESOP)*, *Lecture Notes in Computer Science* 10801, Springer, pp. 999–1026, doi:10.1007/978-3-319-89884-1_35.

[28] Hrutvik Kanabar, Samuel Vivien, Oskar Abrahamsson, Magnus O. Myreen, Michael Norrish, Johannes Åman Pohjola & Riccardo Zanetti (2023): *PureCake: A Verified Compiler for a Lazy Functional Language. Proc. ACM Program. Lang.* 7(PLDI), doi:10.1145/3591259.

[29] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt & Christian Ferdinand (2017): *Closing the Gap – The Formally Verified Optimizing Compiler CompCert.* In: *SSS'17: Safety-critical Systems Symposium 2017*, Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium, CreateSpace, Bristol, United Kingdom, pp. 163–180. Available at `https://inria.hal.science/hal-01399482`.

[30] Steve Klabnik & Carol Nichols (2022): *The Rust Programming Language, 2nd Edition.* No Starch Press. Available at `https://nostarch.com/rust-programming-language-2nd-edition`.

[31] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski & Gernot Heiser (2014): *Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst.* 32(1), doi:10.1145/2560537.

[32] Joomy Korkut (2024): *Foreign Function Verification Through Metaprogramming.* Ph.D. thesis, Princeton University. Currently unpublished thesis draft.

[33] Greg Kroah-Hartman (2005): *Driving Me Nuts - Things You Never Should Do in the Kernel.* Available at `https://www.linuxjournal.com/article/8110`. (Last access: Mar. 18 2024).

[34] Ramana Kumar, Magnus O. Myreen, Michael Norrish & Scott Owens (2014): *CakeML: a verified implementation of ML. SIGPLAN Not.* 49(1), p. 179–191, doi:10.1145/2578855.2535841.

[35] Xavier Leroy (2009): *Formal verification of a realistic compiler. Communications of the ACM* 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[36] Xavier Leroy (2012): *Mechanized Semantics for Compiler Verification.* In: *APLAS 2012 - 10th Asian Symposium on Programming Languages and Systems*, pp. 386–388, doi:10.1007/978-3-642-35182-2_27.

[37] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister & Christian Ferdinand (2016): *CompCert – A Formally Verified Optimizing Compiler.* In: *ERTS 2016: Embedded Real Time Software and Systems*, SEE. Available at `http://xavierleroy.org/publi/erts2016_compcert.pdf`.

[38] Pierre Letouzey (2003): *A New Extraction for Coq.* In Herma Geuvers & Freek Wiedijk, editors: *Types for Proofs and Programs*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 200–219, doi:10.1007/3-540-39185-1_12.

[39] Gregory Michael Malecha (2014): *Extensible Proof Engineering in Intensional Type Theory.* Ph.D. thesis, Harvard University. Available at `https://dash.harvard.edu/bitstream/handle/1/17467172/MALECHA-DISSERTATION-2015.pdf`. (Last access: Mar. 18 2024).

[40] William Mansky, Wolf Honoré & Andrew W. Appel (2020): *Connecting Higher-Order Separation Logic to a First-Order Outside World.* In Peter Müller, editor: *Programming Languages and Systems*, Springer International Publishing, Cham, pp. 428–455, doi:10.1007/978-3-030-44914-8_16.

[41] Yaron Minsky, Anil Madhavapeddy & Jason Hickey (2013): *Real World OCaml: Functional Programming for the Masses.* Real World OCaml, O'Reilly Media.

[42] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock & Dan Grossman (2018): *Œuf: minimizing the Coq extraction TCB.* In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, Association for Computing Machinery, New York, NY, USA, p. 172–185, doi:10.1145/3167089.

[43] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer & Viktor Vafeiadis (2015): *Pilsner: a compositionally verified compiler for a higher-order imperative language.* In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, Association for Computing Machinery, New York, NY, USA, p. 166–178, doi:10.1145/2784731.2784764.

[44] Adriana Nicolae, Paul Irofti & Ioana Leuştean (2024): *OpenBSD Formal Driver Verification with SeL4.* In: *Innovative Security Solutions for Information Technology and Communications: 16th International Conference, SecITC 2023, Bucharest, Romania, November 23–24, 2023, Revised Selected Papers*, Springer-Verlag, Berlin, Heidelberg, p. 144–156, doi:10.1007/978-3-031-52947-4_11.

[45] Tobias Nipkow, Lawrence Charles Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Lecture Notes in Computer Science, Springer Berlin, Heidelberg, doi:10.1007/3-540-45949-9.

[46] Zoe Paraskevopoulou (2020): *Verified optimizations for functional languages.* doi:10.12681/eadd/55614.

[47] Zoe Paraskevopoulou & Andrew W. Appel (2019): *Closure conversion is safe for space. Proc. ACM Program. Lang.* 3(ICFP), doi:10.1145/3341687.

[48] Zoe Paraskevopoulou, John M. Li & Andrew W. Appel (2021): *Compositional optimizations for CertiCoq. Proc. ACM Program. Lang.* 5(ICFP), doi:10.1145/3473591.

[49] Mathieu Paturel, Isitha Subasinghe & Gernot Heiser (2023): *First steps in verifying the seL4 Core Platform.* In: *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '23, Association for Computing Machinery, New York, NY, USA, p. 9–15, doi:10.1145/3609510.3609821.

[50] Christine Paulin-Mohring & Benjamin Werner (1993): *Synthesis of ML programs in the system Coq. J. Symb. Comput.* 15(5–6), p. 607–640, doi:10.1016/S0747-7171(06)80007-6.

[51] John C. Reynolds (2002): *Separation Logic: A Logic for Shared Mutable Data Structures.* In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, IEEE Computer Society, USA, p. 55–74, doi:10.1109/LICS.2002.1029817.

[52] Wang Shengyi (2019): *Mechanized Verification of Graph-Manipulating Programs.* Ph.D. thesis. Available at `https://dl.acm.org/doi/book/10.5555/AAI28828374`. (Last access: Mar. 18 2024).

[53] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau & Théo Winterhalter (2020): *The MetaCoq Project. Journal of Automated Reasoning* 64, pp. 947–999, doi:10.1007/s10817-019-09540-0.

[54] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau & Théo Winterhalter (2019): *Coq Coq correct! verification of type checking and erasure for Coq, in Coq. Proc. ACM Program. Lang.* 4(POPL), doi:10.1145/3371076.

[55] Jr. Guy L. Steele (1978): *Rabbit: A Compiler for Scheme.* Technical Report, MIT, Cambridge, MA, USA. Available at `https://dspace.mit.edu/handle/1721.1/6913`. (Last access: Mar. 18 2024).

[56] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens & Michael Norrish (2016): *A New Verified Compiler Backend for CakeML.* In: *International Conference on Functional Programming (ICFP)*, ACM Press, pp. 60–73, doi:10.1145/2951913.2951924.

[57] Peng Wang, Santiago Cuellar & Adam Chlipala (2014): *Compiler verification meets cross-language linking via data abstraction.* In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &amp; Applications*, OOPSLA '14, Association for Computing Machinery, New York, NY, USA, p. 675–690, doi:10.1145/2660193.2660201.

[58] Shengyi Wang, Qinxiang Cao, Anshuman Mohan & Aquinas Hobor (2019): *Certifying graph-manipulating C programs via localizations within data structures. Proc. ACM Program. Lang.* 3(OOPSLA), doi:10.1145/3360597.

Chapter 14

Reference Manuals and Documentations

[59] Vladimir Brankov (2015): *Inline Assembly for OCaml.* Available at `https://github.com/vbrankov/OCaml-Workshop-2015`. (Last access: Mar. 18 2024).

[60] The kernel development community (2024): *Building External Modules.* Available at `https://www.kernel.org/doc/html/v6.7/kbuild/modules.html`. (Last access: Mar. 18 2024).

[61] The kernel development community (2024): *Command Change Detection.* Available at `https://www.kernel.org/doc/html/v6.7/kbuild/makefiles.html`. (Last access: Mar. 18 2024).

[62] The kernel development community (2024): *Device Drivers.* Available at `https://www.kernel.org/doc/html/v6.7/driver-api/driver-model/driver.html`. (Last access: Mar. 18 2024).

[63] The kernel development community (2024): *Device Power Management Basics.* Available at `https://www.kernel.org/doc/html/v6.7/driver-api/pm/devices.html`. (Last access: Mar. 18 2024).

[64] The kernel development community (2024): *ftrace - Function Tracer.* Available at `https://www.kernel.org/doc/html/v6.7/trace/ftrace.html`. (Last access: June. 06 2024).

[65] The kernel development community (2024): *Input Subsystem.* Available at `https://www.kernel.org/doc/html/v6.7/driver-api/input.html?highlight=input_device`. (Last access: Mar. 18 2024).

[66] The kernel development community (2024): *Kernel Hacking, basic rules.* Available at `https://www.kernel.org/doc/html/v6.7/kernel-hacking/hacking.html#some-basic-rules`. (Last access: Mar. 18 2024).

[67] The kernel development community (2024): *Kernel Stacks.* Available at `https://www.kernel.org/doc/html/v6.7/arch/x86/kernel-stacks.html`. (Last access: Mar. 18 2024).

[68] The kernel development community (2024): *ktime accessors.* Available at `https://www.kernel.org/doc/html/v6.7/core-api/timekeeping.html`. (Last access: June. 06 2024).

[69] The kernel development community (2024): *The Linux Kernel API*. Available at `https://www.kernel.org/doc/html/v6.7/core-api/kernel-api.html`. (Last access: Mar. 18 2024).

[70] The kernel development community (2024): *Memory Allocation Guide*. Available at `https://www.kernel.org/doc/html/v6.7/core-api/memory-allocation.html`. (Last access: Mar. 18 2024).

[71] The kernel development community (2024): *Message logging with printk*. Available at `https://www.kernel.org/doc/html/v6.7/core-api/printk-basics.html`. (Last access: Mar. 18 2024).

[72] The kernel development community (2024): *Platform Device and Drivers*. Available at `https://www.kernel.org/doc/html/v6.7/driver-api/driver-model/platform.html`. (Last access: Mar. 18 2024).

[73] The kernel development community (2024): *The Userspace I/O HOWTO*. Available at `https://www.kernel.org/doc/html/v6.7/driver-api/uio-howto.html`. (Last access: Mar. 18 2024).

[74] The Chicken Scheme Community (2021): *Chicken Scheme Compiler Documentation*. Available at `http://wiki.call-cc.org/man/5/Using%20the%20compiler`. (Last access: Mar. 18 2024).

[75] The Kawa Community (2020): *The Kawa Scheme language*. Available at `https://www.gnu.org/software/kawa/`. (Last access: Mar. 18 2024).

[76] The Scheme Community (2023): *Scheme Implementations*. Available at `https://get.scheme.org/`. (Last access: Mar. 18 2024).

[77] IBM Corporation (2024): *ulimit - Set process limits*. Available at `https://www.ibm.com/docs/en/zos/3.1.0?topic=descriptions-ulimit-set-process-limits`. (Last access: Mar. 18 2024).

[78] Justin Ethier (2024): *Cyclone Scheme*. Available at `https://justinethier.github.io/cyclone/index`. (Last access: Mar. 18 2024).

[79] Justin Ethier (2024): *File Reference for runtime.c*. Available at `http://justinethier.github.io/cyclone/c-api/runtime_8c.html`. (Last access: Mar. 18 2024).

[80] Justin Ethier (2024): *File Reference for types.h*. Available at `http://justinethier.github.io/cyclone/c-api/types_8h_source.html`. (Last access: Mar. 18 2024).

[81] Marc Feeley & contributors (2023): *Gambit Scheme*. Available at `https://gambitscheme.org/latest/`. (Last access: Mar. 18 2024).

[82] Free Software Foundation, Inc (2023): *The GNU make Documentation, v4.4.1, An Introduction to Makefiles*. Available at `https://www.gnu.org/software/make/manual/make.html#Introduction`. (Last access: Mar. 18 2024).

[83] Free Software Foundation, Inc (2023): *Using the GNU Compiler Collection (GCC) v13.2, §4.4*. Available at `https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/Characters-implementation.html`. (Last access: Mar. 18 2024).

[84] Free Software Foundation, Inc (2024): *The GNU C Compiler Collection webpage*. Available at `https://gcc.gnu.org/`. (Last access: Mar. 18 2024).

[85] Free Software Foundation, Inc (2024): *The GNU C Library Reference Manual, v2.39, Consistency Checking.* Available at `https://sourceware.org/glibc/manual/2.39/html_node/Consistency-Checking.html#index-assertions`. (Last access: Mar. 18 2024).

[86] Free Software Foundation, Inc (2024): *The GNU C Library Reference Manual, v2.39, Introduction to Non-Local Exits.* Available at `https://sourceware.org/glibc/manual/2.39/html_node/Non_002dLocal-Intro.html`. (Last access: Mar. 18 2024).

[87] Free Software Foundation, Inc (2024): *The GNU C Library webpage.* Available at `https://sourceware.org/glibc/`. (Last access: Mar. 18 2024).

[88] Free Software Foundation, Inc (2024): *MIT/GNU Scheme.* Available at `https://www.gnu.org/software/mit-scheme/`. (Last access: Mar. 18 2024).

[89] INRIA (2022): *The OCaml Manual, Interfacing C with OCaml.* Available at `https://v2.ocaml.org/releases/4.14/htmlman/intfc.html`. (Last access: Mar. 18 2024).

[90] INRIA (2022): *The OCaml Manual, Native-code compilation (ocamlopt).* Available at `https://v2.ocaml.org/releases/4.14/htmlman/native.html`. (Last access: Mar. 18 2024).

[91] Inria, CNRS and contributors (2021): *The Coq Core Language.* Available at `https://coq.inria.fr/doc/v8.17/refman/language/core/index.html`. (Last access: Mar. 18 2024).

[92] Inria, CNRS and contributors (2021): *The Coq Documentation, Program extraction.* Available at `https://coq.inria.fr/doc/V8.17.0/refman/addendum/extraction.html`. (Last access: Mar. 18 2024).

[93] ISO/IEC (2011): *Committee Draft N1570: Programming languages — C.* Available at `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`. (Last access: Mar. 18 2024).

[94] Michael K. Johnson (1996): *User-space device drivers.* Available at `https://tldp.org/LDP/khg/HyperNews/get/devices/fake.html`. (Last access: Mar. 18 2024).

[95] Jakub Klinkovský (2023): *Broadcom wireless.* Available at `https://wiki.archlinux.org/title/Broadcom_wireless`. (Wiki Entry timestamp: 9 September 2023, at 07:35, Last access: June. 04 2024).

[96] Xavier Leroy, Collège de France & Inria (2023): *CompCert 3.13 Manual.* Available at `https://compcert.org/man/manual003.html#sec23`. (Last access: Mar. 18 2024).

[97] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger & Mark Mitchell (2024): *System V Application Binary Interface AMD64 Architecture Processor Supplement.* Available at `https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/master/raw/x86-64-ABI/abi.pdf?job=build`. (Last access: Mar. 18 2024).

[98] Jonas R. (2021): *Comment on issue about missing .cmd file.* Available at `https://github.com/jbaublitz/knock-out/issues/9#issuecomment-823894583`. (Last access: Mar. 18 2024).

[99] The CertiCoq Team (2024): *The CertiCoq github project page, The CertiCoq plugin.* Available at `https://github.com/CertiCoq/certicoq/wiki/The-CertiCoq-plugin`. (Last access: Mar. 18 2024).

[100] The GHC Team (2023): *Runtime system (RTS) options.* Available at `https://downloads.haskell.org/ghc/9.8.1/docs/users_guide/runtime_control.html`. (Last access: Mar. 18 2024).

[101] The GHC Team (2023): *Runtime system (RTS) sources.* Available at `https://gitlab.haskell.org/ghc/ghc/-/blob/ghc-9.8.1-release/rts/Weak.c?ref_type=tags`. (Last access: Mar. 18 2024).

[102] The uClibc-ng Team (2024): *The uClibc-ng webpage.* Available at `https://uclibc-ng.org/`. (Last access: Mar. 18 2024).

[103] The Clang Team (2024): *Clang: a C language family frontend for LLVM.* Available at `https://clang.llvm.org/`. (Last access: Mar. 18 2024).

[104] Linux Torwalds (2004): *Re: How to use floating point in a module?* Available at `https://yarchive.net/comp/linux/kernel_fp.html`. (Last access: Mar. 18 2024).

[105] Linux Torwalds (2016): *BUG_ON() in workingset_node_shadows_dec() triggers.* Available at `https://lkml.org/lkml/2016/10/4/1`. (Last access: Mar. 18 2024).

[106] Andrew Waterman & Krste Asanović (2019): *The RISC-V Instruction Set Manual.* Available at `https://riscv.org/specifications/`. (Last access: Mar. 18 2024).

Chapter 15

Software Artifacts

[107] Xia Li yao et al (2024): *Examples folder of the VeriFFI github project page.* Available at `https://github.com/Lysxia/coq-simple-io`. (Last access: June 04 2024).

[108] bootlin (2024): *Elixir Cross Referencer, longjmp identifier.* Available at `https://elixir.bootlin.com/linux/v6.7/A/ident/longjmp`. (Last access: Mar. 18 2024).

[109] The kernel development community (2017): *The Linux Kernel Sources, definition of EINVAL.* Available at `https://github.com/torvalds/linux/blob/v6.7/include/uapi/asm-generic/errno-base.h#L26`. (Last access: Mar. 18 2024).

[110] The kernel development community (2017): *The Linux Kernel Sources, Log Levels Header File.* Available at `https://github.com/torvalds/linux/blob/v6.7/tools/include/linux/kern_levels.h`. (Last access: Mar. 18 2024).

[111] The kernel development community (2021): *The Linux Kernel Sources, setjmp_64.S.* Available at `https://github.com/torvalds/linux/blob/v6.7/arch/x86/um/setjmp_64.S`. (Last access: Mar. 18 2024).

[112] The kernel development community (2022): *The Linux Kernel Sources, definition of EV_SND.* Available at `https://github.com/torvalds/linux/blob/v6.7/include/uapi/linux/input-event-codes.h#L45`. (Last access: Mar. 18 2024).

[113] The kernel development community (2023): *The Linux Kernel Sources, Bug Header File.* Available at `https://github.com/torvalds/linux/blob/v6.7/include/asm-generic/bug.h`. (Last access: Mar. 18 2024).

[114] The kernel development community (2023): *The Linux Kernel Sources, exit.c.* Available at `https://github.com/torvalds/linux/blob/v6.7/kernel/exit.c#L809`. (Last access: Mar. 18 2024).

[115] The kernel development community (2023): *The Linux Kernel Sources, IO.* Available at `https://github.com/torvalds/linux/blob/v6.7/include/asm-generic/io.h`. (Last access: Mar. 18 2024).

[116] The kernel development community (2023): *The Linux Kernel Sources, PC Speaker Driver.* Available at `https://github.com/torvalds/linux/blob/v6.7/drivers/input/misc/pcspkr.c`. (Last access: Mar. 18 2024).

[117] The kernel development community (2023): *The Linux Kernel Sources, spinlocking.* Available at `https://github.com/torvalds/linux/blob/v6.7/include/linux/spinlock.h`. (Last access: Mar. 18 2024).

[118] The kernel development community (2024): *The Linux Kernel Sources, Kernel Makefile.* Available at `https://github.com/torvalds/linux/blob/v6.7/Makefile`. (Last access: Mar. 18 2024).

[119] The OCaml community (2020): *OCaml misc.c Sources.* Available at `https://github.com/ocaml/ocaml/blob/4.14.1/runtime/misc.c`. (Last access: Mar. 18 2024).

[120] The OCaml community (2021): *OCaml floats.c Sources.* Available at `https://github.com/ocaml/ocaml/blob/4.14.1/runtime/floats.c`. (Last access: Mar. 18 2024).

[121] The OCaml community (2021): *OCaml memory.c Sources.* Available at `https://github.com/ocaml/ocaml/blob/4.14.1/runtime/memory.c`. (Last access: Mar. 18 2024).

[122] The OCaml community (2021): *OCaml memory.c Sources.* Available at `https://github.com/ocaml/ocaml/blob/4.14.1/runtime/major_gc.c`. (Last access: Mar. 18 2024).

[123] The OCaml community (2021): *OCaml mlvalues Sources.* Available at `https://github.com/ocaml/ocaml/blob/4.14.1/runtime/caml/mlvalues.h`. (Last access: Mar. 18 2024).

[124] The OCaml community (2021): *OCaml sys.c Sources.* Available at `https://github.com/ocaml/ocaml/blob/4.14.1/runtime/sys.c`. (Last access: Mar. 18 2024).

[125] The CertiGraph Project Contributors (2024): *The CertiGraph Github Repository.* Available at `https://github.com/CertiGraph/CertiGraph`. (Last access: Mar. 18 2024).

[126] The Œuf Project Contributors (2019): *The Œuf Github Repository.* Available at `https://github.com/uwplse/oeuf`. (Last access: Mar. 18 2024).

[127] Mario Frank (2024): *A Partial Formalisation of the Linux PC Speaker Driver*, doi:10.5281/zenodo.10715853.

[128] Mario Frank (2024): *Adoption of CertiCoq runtime for compatibility with Linux Kernel Modules.* Available at `https://github.com/eladrion/certicoq/tree/fphd`. (Last access: Mar. 18 2024).

[129] Mario Frank (2024): *Adoption of VeriFFFI functionality for compatibility with Linux Kernel Modules.* Available at `https://github.com/eladrion/VeriFFI/tree/f60c69f3ab8afe2a6755d345f42b46b45c043f01`. (Last access: Mar. 18 2024).

[130] Mario Frank (2024): *Extend the Extraction Plugin to synthesise OCaml external and callback definitions for interfacing C/C++.* Available at `https://github.com/coq/coq/pull/18270/`. (Last access: Mar. 18 2024).

[131] Mario Frank (2024): *The (experimental and partially) Certified Linux PC Speaker Driver*, doi:10.5281/zenodo.10707318.

[132] Mario Frank, Mario Egger & Andreas Otto (2023): *The OCaml Subsystem for L4Re and the Cross ocamlopt OPAM packages repo*, doi:10.5281/zenodo.10192040.

[133] Kernkonzept GmbH (2023): *The L4Re Operating System Wiki.* Available at `https://github.com/kernkonzept/manifest/wiki`. (Last access: Mar. 18 2024).

[134] Kernkonzept GmbH (2024): *The L4Re Microkernel Repository.* Available at `https://github.com/kernkonzept/fiasco`. (Last access: Mar. 18 2024).

[135] Kernkonzept GmbH (2024): *L4Re NVMe server.* Available at `https://github.com/kernkonzept/nvme-driver`. (Last access: Mar. 18 2024).

[136] Johnathan Nightingale, Chris Wong, Rhonda D'Vine & Jérôme Lafréchoux (2013): *beep.* Available at `https://github.com/johnath/beep`. (Last access: June 06 2024).

[137] Raster Software, Vigo (2018): *An user-space driver for Silead's GSL1680 capacitive touch screen driver chip.* Available at `https://github.com/rastersoft/gsl1680`. (Last access: Mar. 18 2024).

[138] Manuel Serrano (2022): *The Bigloo Scheme Sources, cerror.c.* Available at `https://github.com/manuel-serrano/bigloo/blob/4.5b/runtime/Clib/cerror.c`. (Last access: Mar. 18 2024).

[139] Manuel Serrano (2024): *The Bigloo Scheme Sources.* Available at `https://github.com/manuel-serrano/bigloo`. (Last access: Mar. 18 2024).

[140] Cisco Systems (2024): *The Chez Scheme Sources.* Available at `https://github.com/cisco/ChezScheme`. (Last access: Mar. 18 2024).

[141] The CakeML Team (2024): *CakeML: A Verified Implementation of ML.* Available at `https://github.com/CakeML/cakeml`. (Last access: Mar. 18 2024).

[142] The CertiCoq Team (2024): *The CertiCoq github project page, Plugin runtime.* Available at `https://github.com/CertiCoq/certicoq/tree/master/plugin/runtime`. (Last access: Mar. 18 2024).

[143] The VeriFFI Team (2024): *Examples folder of the VeriFFI github project page.* Available at `https://github.com/CertiCoq/VeriFFI/blob/main/examples/`. (Last access: Mar. 18 2024).

[144] The VeriFFI Team (2024): *The VeriFFI github project page.* Available at `https://github.com/CertiCoq/VeriFFI`. (Last access: Mar. 18 2024).

[145] Gwen Weinholt (2024): *Loko Scheme.* Available at `https://gitlab.com/weinholt/loko`. (Last access: Mar. 18 2024).

Acknowledgements

I would like to express my gratitude to all those who have provided me with invaluable support. Firstly, I would like to thank my family for their unwavering support throughout the years. The journey towards completing my PhD thesis was not always straightforward, and I am grateful that I was able to finish it thanks to your encouragement. I love you all.

Secondly, I would like to express my appreciation to Prof. Dr. Christoph Kreitz, who saw potential in me even though I was a relatively average student. Furthermore, I would like to express my gratitude to the current and former members of the theoretical computer science chair, in particular Dr. Eva Richter, Tim Richter, Dr. Sebastian Böhne, Nuria Brede and Dr. Jens Otten. Our numerous formal discussions undoubtedly contributed to the development of my reasoning abilities. Additionally, we enjoyed fruitful collaboration in both scientific and teaching endeavours.

Furthermore, I would like to express my gratitude to the numerous reviewers, both of scientific contributions and software projects, whose input has contributed to the quality of these works. In particular, I would like to thank the reviewers of this thesis for identifying areas for improvement, which have enabled me to enhance the thesis.