Bringing Synthesised Software to a Real-World Microkernel Operating System

Mario Frank
Mario Egger
Christoph Kreitz
mario.frank@uni-potsdam.de
mario.egger@uni-potsdam.de
christoph.kreitz@uni-potsdam.de
University of Potsdam, Institute of Computer Science
Potsdam, Brandenburg, Germany

Andreas Otto
Kernkonzept GmbH
Dresden, Saxony, Germany
andreas.otto@kernkonzept.com

ABSTRACT

We present the package *OSL4*, that is able to cross compile OCaml source code for the L4Re Operating System Framework, currently unsupported by upstream OCaml. We show the necessity of this kind of compilation, originating from the concept of a minimal trusted computing base (TCB), and how the package can be used to make verified software and OCaml software in general executable in a system for highly security critical applications. We analyse existing cross compilation solutions concerning the usefulness for our project aims. Then we analyse the general OCaml project structure and compilation pattern to identify a general approach for building OCaml cross compilers.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Functional languages**; • **Security and privacy** → *Operating systems security*; Logic and verification.

KEYWORDS

Cross Compilation, Interpreter, OCaml, L4Re

ACM Reference Format:
Mario Frank, Mario Egger, Christoph Kreitz, and Andreas Otto. 2023. Bringing Synthesised Software to a Real-World Microkernel Operating System. In *Proceedings of The 35th Symposium on Implementation and Application of Functional Languages (IFL'23).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Security critical infrastructure, like automotive systems and governmental IT environments are a constant target of attacks. Thus, operating systems should have a minimum level of security to be viable for use in those areas. But the state of the art systems do look quite different. For example, Cloud infrastructure is usually built upon common operating systems like Linux[9], BSD[4] or Microsoft Windows[10]. In order to be viable for use in security critical applications, relevant software - including operating systems - has to be certified[4] in some countries. This is done, for example, by the BSI in Germany.

A certification usually involves:

- Code audits
- Documentation audits
- Documenting processes and audits of these documents
- Third party (penetration) tests
- Site-security inspections

But this certification can never be sufficient to assess the security of a general purpose operating system in full depth. The Linux Kernel, for example, consists of 30 M[15] lines of code. Auditing the complete code would take an unfeasibly long time during which the system will have changed significantly.

In order to be able to do a reasonable audit, the trusted computing base (TCB), i.e. the minimal set of code that is necessary to provide the desired functionality, has to be as small as possible. But even then, a code audit will not catch every dysfunction or logical error in the software. The use of static analysis tools, like Coverty[11] or CppCheck[19], does reduce the count of potential bugs - but still is not a complete solution to the problem.

This is where formal methods come into play. Provided a formal specification of, e.g., the OS functionality, security properties can be proven. Also, a specification of the properties in proof assistants like Coq[18] can be used to extract the concrete function, i.e. synthesise software source code.

But in an industrial setting, using that synthesised software is not as easy as it may seem. A company, providing a fully functional OS will need either a massive amount of additional, qualified employees or a large time span in order to completely verify or replace it with a fully synthesised variant. A solution can be to replace the existing OS functionality gradually. But in order to accomplish this, the synthesised portions need to be compatible with the existing system. More specifically, a toolchain that compiles the synthesised code for the target system and architecture must exist. This work describes the attempt of creating such a toolchain for the L4Re Operating System Framework.

IFL'23, August 29–31, 2023, Braga, Portugal
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 L4RE AND THE VERIFIED SECURE CLOUD RESEARCH PROJECT

The L4Re Operating System Framework[7] provides the L4Re microkernel and user-space components from which operating systems and hypervisors for diverse applications can be assembled. True to the microkernel philosophy, the L4Re microkernel only implements the basic functionality to manage the hardware and run applications. All other functionality is provided by dedicated applications, for instance:

- The Moe root task provides process management and memory allocation to all other applications.
- The Io server handles all hardware devices and grants access to their I/O memory, ports and interrupts to other applications on request.
- The Ned init process executes a Lua script to setup the system to the configurator's specification.
- The Uvmm virtual machine monitor runs guest operating systems.
- Different programs are device drivers for, e.g., NVMe and AHCI drives, eMMC storage, and VirtIO devices.

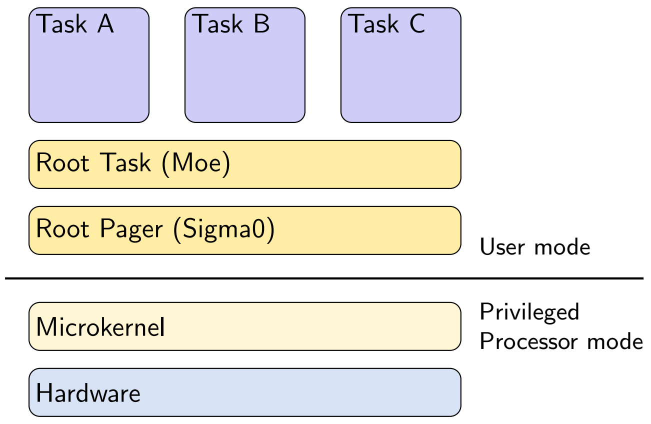

Figure 1: The basic structure of a typical L4Re system.

Figure 1 shows an illustration of this separation of functionality. This approach minimizes the TCB. In particular, only the microkernel has to run in the processor's highest privilege level and contains very little code in comparison to monolithic kernels. The microkernel implements a capability system in which access to memory and kernel objects (such as threads, tasks, and interrupts) is mediated via access tokens called capabilities. This ensures resource isolation and non-interference between applications. Moreover, capabilities can be passed from application to application to enable resource transfer or sharing. The Io server for example uses this to grant access to a device's I/O memory to the device driver application.

The Verified Secure Cloud Research Project (VerSeCloud[6, 14]) intends to enable L4Re to serve as a cloud hypervisor for security critical workloads. The project comprises three main areas:

(1) Improving the L4Re infrastructure, in particular the Uvmm virtual machine monitor, to add needed cloud functionality.
(2) Using formal methods and model based testing to verify L4Re for security critical contexts.
(3) Developing a framework for generating verified, secure device drivers for L4Re.

For the last point, the Kernkonzept GmbH collaborates with the University of Potsdam.

As described, drivers are implemented as userspace packages in contrast to monolithic Operating Systems, which realize peripheral communication through kernel modules. Moreover, in order for L4Re to be usable as a cloud hypervisor, more device drivers are necessary. The most vital drivers for cloud infrastructures are the network (ethernet) and data storage (block device) drivers. The aims of the project division at the University of Potsdam are to formalise device driver functionality, verify properties like termination and correctness of outputs for respective inputs. And most importantly, the overall result shall be the synthesis of device driver components that are executable in L4Re.

Since vendors of device drivers seldom expose specifications of their device drivers existing device driver source code has to be analysed. This way, we extract the functional properties of the device driver source code. With this knowledge, we formalise those functions and their properties in the proof assistant Coq. After verifying that the stated properties do hold for all possible inputs in the input domain, we are able to synthesise functional source code. During this process all definitions and theorems are collected in a formalisation library and gradually generalised in order to be reusable for the formalisation of the functionality of other drivers. As a matter of fact, device driver source code often contains parts that are not free of side effects, for example when writing to registers, which does not necessarily give return values. Thus, it is hardly possible to formalise all aspects of a device driver. But the functions containing side effects are usually the ones, where the driver communicates with the operating system itself. Thus, we have to introduce assumptions about what will be the result of calling a function of the L4Re microkernel or apply an axiomatic definition. Although a complete verification is not possible under those circumstances, the results will improve the security of and trust in the drivers a lot.

3 ENRICHING EXISTING SYSTEMS WITH SYNTHESISED SOFTWARE

As described, the Coq Proof Assistant provides functionality to extract functional source code from proofs and definitions written in Coq. These extraction targets the two functional languages OCaml and Haskell natively as well as a subset of the C programming language called Clight[2] using the Coq plugin CertiCoq[1].

In the following, we describe the synthesis options and discuss their advantages and drawbacks. Also we show the necessity of creating a cross compilation solution of OCaml for L4Re.

3.1 Low Level Code Synthesis

As already noted, Clight is a real subset of the C programming language and can be extracted from Proofs with the Coq module CertiCoq. The generated C code can be compiled with the normal GNU C Compiler. Additionally, the formally verified compiler CompCert[12] can be used to compile Clight. Using this extraction and compilation toolchain would have the huge advantage that there would be a consistent chain of trust in our toolchain as it would be all verified tools up until the code execution. However, the C code generated by CertiCoq contains dependencies to Coq, i.e. the normal use case is to execute the compiled program in Coq. This sadly does not fit into our use case. A solution would be to either provide a special wrapper for the generated code which would break the chain of trust. Another solution would be to adopt the generation algorithm of CompCert to create standalone code. But this is outside the research project scope - both in effort and in time.

3.2 Functional Extraction

The other option would be to extract the definitions and proof objects from Coq into one of the functional languages Haskell or OCaml. However, we lack a compiler both for Haskell and OCaml to translate the extracted language into interpretable bytecode or executable machine code for L4Re. The question here is, whether it would be possible to create such a compiler. There is no native C compiler for L4Re, i.e. no C compiler that can be executed in a running L4Re instance. Instead, all C and C++ software is compiled on some development machine for running under L4Re. So, in fact, software is cross compiled for L4Re.

The Glasgow Haskell Compiler (GHC) is written in Haskell and building the GHC for a given system requires an already compiled GHC together with additional extensions.[21] But there is no such GHC for L4Re. Building a cross compiler also requires a prebuilt GHC and additional system headers for the host system.[20] So in principle, it should be possible to create a cross compiler for Haskell as there does exist a cross compilation documentation for the Raspberry Pi. But creating a GHC cross compiler is quite cumbersome due to the dependency on an existing GHC[27][13].

Building OCaml for a specific system usually requires an existing C compiler toolchain in the running target system.[25] Thus, it is not possible to build an OCaml compiler for L4Re without porting the C compiler for L4Re which is out of project scope. But OCaml has the advantage compared to Haskell, that it ships a prebuilt bytecode variant of the OCaml compiler. Thus, only a C compiler is necessary to create a fully working OCaml compiler suite, i.e. an OCaml interpreter (*ocamlrun*) that can interpret the shipped OCaml compiler. So potentially, it should be easier to creating a working cross compilation solution than for Haskell. Another reason to prefer OCaml over Haskell is the obvious syntactic and semantic proximity of Coq to OCaml as Coq is a OCaml module and, more important, implemented in OCaml itself. Moreover, the OCaml extraction in Coq seems to be maintained better than the Haskell extraction - but this is mainly a subjective argument.

These aspects considered we decided to target OCaml as the language for code synthesis. This being said, a cross compilation of OCaml for L4Re seems to be both in project scope and timely achievable.

3.3 Existing Cross Compilation Approaches

There have been numerous attempts on creating an OCaml cross compiler. Two cross compilation schemes are obvious use cases, cross compiling for a different architecture (e.g. x86 to ARM) or for a different operating system (e.g. Linux to Windows/macOS).

An approach for the former, i.e. cross compiling OCaml on x86 for ARM with Linux as OS, is the *addnas_ocaml-cross-compiler*[29] project. But it is limited to OCaml 3.12 and discontinued for more than 10 years which disqualifies it as starting point for a solution.

The ReasonML Mobile project[16] aims at both cross compiling for a different architecture and operating system. But it is limited to Android and iOS mobile devices as target and adopted to the esy[3] package manager which is not compatible with the L4Re toolchain. Adding esy as dependency is furthermore far too much overhead for our use case.

The most up to date and sophisticated cross compiler collections are OCaml-Cross[30] and the cross compiler for MirageOS[22]. OCaml-Cross supports cross compilation both for different operating systems (Linux or macOS to Windows, macOS to iOS and Linux to Android) and different architectures. But the latter is limited to compiling on a Linux x86 machine to Android ARM (32 and 64 bits). Although this suite is maintained very well it is not suitable for our purposes, since the project uses the OCaml package management (OPAM) which does not fit well into the toolchain of L4Re. Also, the cross compiler patches the original OCaml project sources which we want to avoid as it would make the OCaml dependency a moving target and increase the maintenance overhead significantly.

MirageOS is an operating system implemented as library. The project provides an OCaml compiler that can be used to create a standalone unikernel, executing the functionality of the OCaml software being compiled. The software is bundled together with the necessary kernel components as binary code and can be deployed in a virtualisation solution. However, for cross compilation, the MirageOS toolchain uses the solo5 cross compiler[23]. The cross compiler is specifically targeting solo5[24] by providing a customised runtime for the solo5 architecture which makes it incompatible with L4Re. Thus, this cross compiler is not applicable to our use case without significant adoptions.

There are also other still maintained approaches, like OCaPIC[17]. which was extended by the OMicroB[28] team in order to provide an OCaml virtual machine for OCaml programs and uses OCaPIC. But both of them aim at programming microcontrollers with OCaml which is too far away from our use case. To sum up, the available solutions are not applicable to our use case.

4 BUILDING A MINIMAL OCAML CROSS COMPILER

In order to be able to compile synthesised OCaml code for L4Re, we need to use an appropriate OCaml compiler. However, we found no straight-forward way to integrate the upstream OCaml compiler with L4Re build system. In the following, we give a brief overview about the OCaml project structure and the way the OCaml compiler works. Then we describe the necessary steps to create a suitable OCaml compiler that matches our desired functionality.

4.1 About the OCaml Project Structure and Components

The OCaml compiler suite[26] is a bootstrapping based project. The basis is built upon an already provided bytecode compiler (*ocamlc*) and the runtime library, that is implemented in C. The host C compiler is used to compile the low-level runtime (*libasmrun.a*), the interpreter runtime (*camlrun.a*) and the OCaml interpreter (*ocamlrun*). Then, *ocamlrun* is used to interpret the provided *ocamlc* and compile a new *ocamlc*, the bytecode standard library and the native code compiler (*ocamlopt*). The native code compiler itself is afterwards used to compile the native code standard library, a standalone bytecode (*ocamlc.opt*) and native (*ocamlopt.opt*) compiler. There are many components built during this process, like *ocamldep*, but those are not of interest for our toolchain.

The target architecture and system on which the compiler shall run and for which the compiled programs shall be suitable, are defined by two main files: The machine (*m.h*) and operating system (*s.h*) functionality specification header files. Both files are generated during the configuration step of the OCaml project compilation, depending on the chosen parameters (for *build*, *host* and *target*). For OCaml 4.13, a multitude of target architectures are supported, including *x86*, *ARM*, and *Power PC* (each 32 and 64 bits). Also, depending on the architecture, OCaml can be compiled to run on multiple operating systems, like Linux, Microsoft Windows, macOS and different BSD variants.

But in all cases, OCaml is expected to run on the target operating system and architecture. The main problem here is, that the target operating system needs to have a working C compiler toolchain in order to do a native code compilation with *ocamlopt* since *ocamlopt* itself uses the specified C compiler. While this does hold for the named supported operating systems, it does not for L4Re, as already noted. While porting a C compiler for L4Re may be possible, it is out of scope of the research project.

Thus, there is a need for an OCaml cross compiler, running on standard hardware (*x86*), that can produce native code for L4Re for a concrete set of target architectures (*x86* and *ARM* for the scope of the research project). Furthermore, for this kind of application, we can restrict the set of components to handle to the minimum, i.e. the low-level runtime (*libasmrun.a*) for linking the partially compiled binaries together with the L4Re libraries, the native code compiler (*ocamlopt*) and the native standard library. As a side effect of the toolchain that will be described in the following sections, we also gain an OCaml interpreter (*ocamlrun*) and a bytecode standard library, both working in L4Re.

4.2 Leveraging the OCaml Compilation Process

For producing a cross compiler, we leverage the compilation process of *ocamlc* and *ocamlopt*, as illustrated in Figure 2. Considering a standalone OCaml file, i.e. a file that includes a main function and only has dependencies to the OCaml standard library, there are two ways of compiling that file. First, we can use the host *ocamlc* to compile the OCaml file into an OCaml bytecode object. Then, we only need the bytecode standard library (*stdlib.cma*) and the bytecode interpreter *ocamlrun*. While the *stdlib.cma* is platform and OS independent, *ocamlrun* has to be compiled specifically for L4Re.

For native code compilation, more work has to be done. As L4Re does not contain a C Compiler, the compilation has to be done under Linux, for example. But *ocamlopt* does help us here. This compiler itself leverages the functionality of the host C compiler, i.e. *ocamlopt* is able to produce a C object which is then linked to a binary executable using the GNU C Compiler, for example. During this process of compilation and linking, *ocamlopt* sets the concrete C compiler, its options and which libraries to link. By default, *ocamlopt* signals to link the OCaml runtime (*libasmrun.a*)

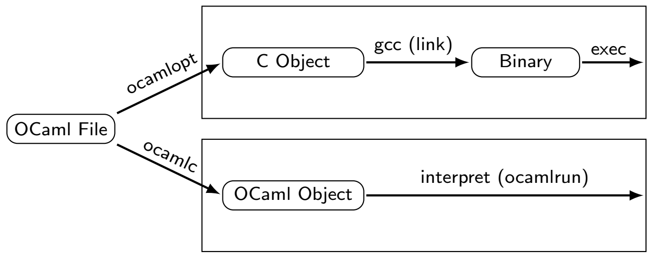

Figure 2: Variants of compiling OCaml files.

and the native code standard library (*stdlib.a*). Thus, in principle we "merely" have to override the call to the C compiler and also provide L4Re specific libraries. Technically, this is more complex than it may seem since it is not possible to build the *stdlib.a* without previously building the *ocamlopt* compiler. Thus, we also cannot just use an installed OCaml compiler and feed it with specialised libraries. But from this analysis, we get a quite concrete view of the requirements to build such a cross compilation toolchain, which we show in Table 1.

Table 1: Necessary components for the host system, bytecode execution in and native compilation for L4Re

Component	Host	Byte@L4Re	Native for L4Re
libasmrun.a	✓ (x86)	X	✓
ocamlrun	✓ (x86)	✓	X
ocamlc	✓	X	X
ocamlopt	✓	X	X
stdlib.cma	✓	✓	X
stdlib.a	X	X	✓

4.3 Establishing a Cross Compilation Toolchain

Although OCaml is not targeting cross compilation explicitly, the OCaml project compilation toolchain (Makefiles) has a structure that can be used quite elegant to build a cross compiler. For simplicity, we describe the process for the target architecture *aarch64* while the host architecture is *x86_64* in our use case.

According to the previous descriptions, we have three tasks:

(1) Build the L4Re specific *aarch64* runtime and interpreter
(2) Build an OCaml *x86_64* to *aarch64* cross compiler
(3) Build the *aarch64 stdlib.a*

The good thing is, that the *stdlib.a* can be produced as side effect of step 2. And both for step 1 and 2, we need a clean distribution of the OCaml project. In the following, we describe the process of creating the cross compiler, as shown in Figure 3 where the diamond shaped boxes represent splits and the empty bars represent the fusion of workflows.

4.3.1 Building the L4Re specific aarch64 runtime and interpreter. For building the L4Re specific runtime and interpreter, the distribution is configured with host and target being *aarch64*. The configure script then automatically generates all architecture specific files, most important the *m.h*. Then, a custom Makefile produces the

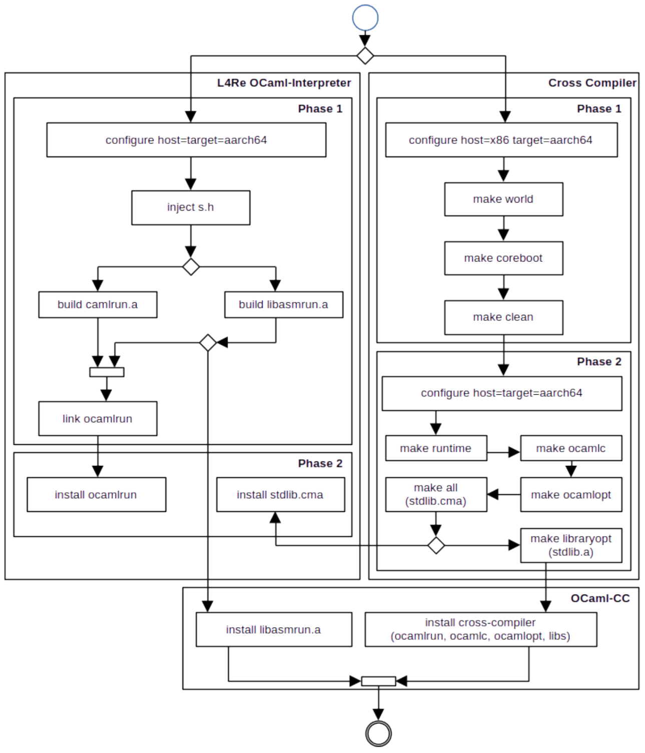

Figure 3: The process of producing an OCaml cross compiler for L4Re.

libasmrun.a, *libcamlrun.a* and *ocamlrun* for L4Re. This is done using the *aarch64* C cross compiler with options for adopting the output objects and binaries to L4Re. All those files are then copied or linked in the L4Re build tree to be reusable in later steps. One crucial part is overriding the *s.h* that was generated by the configure script with a custom set of defines that adhere to the functionality of L4Re. But this process has the advantage that no change in the original OCaml project code is necessary.

4.3.2 Building an OCaml x86_64 to aarch64 cross compiler. For this step, a clean copy of the OCaml project is first configured to create a compiler running on the host *x86_64*, producing code for target *aarch64*. Then, we can create the runtime and bytecode interpreter using the **world** target. This way, we get an OCaml interpreter running on our *x86_64* machine, i.e. on Linux. The generated interpreter and libraries are then set as our default interpreter for the next steps by calling the target **coreboot**. This target makes sure, that all necessary components are copied in the special boot folder of the OCaml distribution.

Following this, we need to create the compilers and also the native standard library for *aarch64*. This is done by cleaning the project tree while preserving the interpreter and then configuring the project for both *host* and *target* being *aarch64*. Then, we can compile the runtime and the bytecode variants of *ocamlc* and *ocamlopt*. Those files will happily run in the interpreter produced previously, but generate our desired *aarch64* C object code.

As final step, we need to produce the OCaml standard library - both the bytecode and the native variant. This can be done quite easily. The bytecode standard library is produced by calling the target **all**, while the target **libraryopt** produces the native code standard library in the form of a *stdlib.cmxa* and *stdlib.a*. The latter one then can be used in the linking phase of an arbitrary C compiler.

The resulting binaries, bytecode objects and libraries are then locally installed into a special cross compiler folder. Also, we inject the L4Re specific *libasmrun.a* that was produced in step 1 into this folder, giving us a standalone OCaml cross compiler for L4Re.

4.4 Trustworthiness

But there is something we have to assess - the trustworthiness of our solution. If we consider the compilation of synthesised, correct code: Is it possible that our compilation pattern breaks the chain of trust? To answer this question, we recall the possible ways to compile synthesised code, shown in Figure 4. Trusted steps are represented by solid arrows, while not (fully) trustworthy steps are represented by dotted arrows. Rectangles with sharp corners represent tools and rectangles with rounded corners represent files.

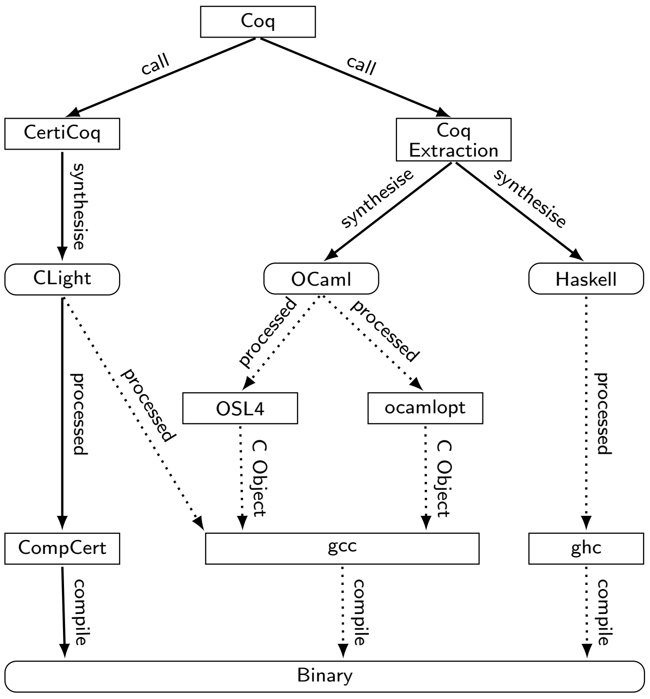

Figure 4: The Chain of Trust when compiling synthesised code.

We can, and have to, trust the extractions by CertiCoq and the Coq Extraction mechanism since both are formalised in Coq. Thus, the generated Clight, OCaml and Haskell code are trustworthy. Also, we can trust CompCert to correctly generate the binary code since it was proven to be correct. But when it comes to compiling OCaml or Haskell code, the chain of trust breaks. Neither *ocamlopt*, nor the GHC are verified to compile correctly. Also, *ocamlopt* uses the gcc which is not verified. Since *OSL4* does not modify the OCaml compiler but only sets the functionality supported by the OS by defining the *s.h*, the compilation done by *OSL4* is identical to the point where the GCC is called. Thus, the binary compiled via *OSL4* is neither more nor less trustworthy than the default compilation by OCaml. Apart from that, if we do not trust *ocamlopt*, we cannot even trust the binary version of Coq which is implemented in OCaml itself.

5 USING THE OSL4 L4RE-PACKAGE

All steps described in the previous section are bundled as the OCaml Subsystem for L4Re (*OSL4*). This bundle is a regular L4Re package. L4Re packages contain a definition file *Control* that specifies

- Dependencies of the package (tag *requires*)
- Provided functionality (tag *provides*)
- The maintainer

Using those information, the L4Re toolchain is able to resolve dependencies and make sure that all required packages are built before being used.

OSL4 itself contains 2 files being exposed to other packages which are a Makefile for OCaml bytecode compilation and OCaml native cross compilation.

Using OSL4 for Bytecode Compilation. After including the *ml-build_byte.mk*, all OCaml files being in the source folder of a package using *OSL4*, are automatically compiled to OCaml object files using the generated *ocamlc* compiler. The using package then has to make sure that the bytecode files are installed into the L4Re build tree but that is an easy task when using the L4Re Makefile toolchain.

Using OSL4 for Native Code Compilation. Before including the *mlbuild_native.mk*, the using package has to specify a main OCaml file (variable *ML_MAIN_FILE*) which is the file that contains the main routine of the OCaml tool. Also additional OCaml files can be specified as dependencies during build (variable *ML_DEP_FILES*). All specified files are then automatically compiled to C object files and then to a binary executable by the L4Re toolchain.

It is also possible to generate a C object library that can communicate via the OCaml foreign function interface with the L4Re infrastructure.

6 STATUS, LIMITATIONS AND FUTURE WORK

Currently, OSL4 supports the compilation of OCaml to bytecode objects that can run under L4Re. Also, it is possible to compile OCaml code to native L4Re executable files for *arm*, *aarch64* and *x86_64*. We were able to compile an example OCaml tool for reading and printing the contents of an ISO9660 file in a running L4Re environment. This tool was mostly implemented in OCaml but uses the OCaml foreign function interface with C to get the necessary capabilities from L4Re to get access to the ISO file. Thus, we can not only compile standalone command line tools, but also components that interact with L4Re. For our project scope, this set of features is currently sufficient. Nevertheless, there is much what can be improved.

For example, it is currently not possible to produce shared C objects, i.e. dynamic linking is currently not possible. This is definitely desirable but currently out of scope of the project. Moreover, OSL4 currently supports OCaml 4.13 and 4.14 - Adding support for OCaml 5 would give the opportunity to produce multithreaded code. Additionally, the target architectures are more limited than what L4Re supports. Adding support for those architectures, like MIPS, Sparc, or RISC-V would be a great improvement. Also, no additional OCaml packages are ported for L4Re since they are not necessary for our project aims.

Finally, although OSL4 is built for L4Re, the package could be opened for different operating systems by providing a new OPAM package, potentially on the basis of OCaml-Cross, that is then used by OSL4. This would eliminate the direct dependency to OCaml Project sources in L4Re. Another approach, that would be even better: If only the runtime is OS specific, our concept could be generalised. This could be done by creating cross compilers for different target architectures with a clear naming convention (as common for GCC), e.g.

- *x86_64_ocamlopt*
- *arm_ocamlopt*
- *aarch64_ocamlopt*

⋮

and provide them as OPAM package without being usable standalone. More specific, the compiler should be (statically) built to run on the host machine, but the installation process should omit the *libasmrun.a*. Then, for each target operating system where cross compilation is needed, an OPAM package could be created, that depends on the architecture specific OPAM package. That one could build the specialised runtime for that system and architectures, (e.g. *aarch64_l4re_ocamlopt*) that automatically calls the appropriate compiler and sets the runtime location when calling the compiler.

ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research under the VerSeCloud project (16KIS1357K). Special thanks go to Steffen Liebergeld for internally reviewing this work and giving helpful comments.

REFERENCES

[1] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *The third international workshop on Coq for programming languages (CoqPL)*. Retrieved July 31, 2023 from https://certicoq.org/

[2] Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288. http://xavierleroy.org/publi/Clight.pdf

[3] The esy Project Team. 2023. *esy - Easy package management for native Reason, OCaml and more*. Retrieved July 31, 2023 from https://github.com/esy/esy

[4] Federal Office for Information Security. 2023. The value of information security: certification and approval by the BSI. Retrieved July 31, 2023 from https://www.bsi.bund.de/EN/Themen/Unternehmen-und-Organisationen/Standards-und-Zertifizierung/Zertifizierung-und-Anerkennung/zertifizierung-und-anerkennung_node.html

[5] Mario Frank and Mario Egger. 2023. *OCaml Subsystem for L4Re*. Retrieved July 31, 2023 from https://gitup.uni-potsdam.de/versecloud-up/osl4

[6] Nicolas Frinker, Steffen Liebergeld, Dr. Andreas Otto, Mario Frank, and 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, 227–240.

[7] Kernkonzept GmbH. 2023. The L4 Operating System Framework. Retrieved July 31, 2023 from https://www.kernkonzept.com/l4re-operating-system-framework/

[8] Kernkonzept GmbH. 2023. *The L4Re Operating System Wiki*. Retrieved July 31, 2023 from https://github.com/kernkonzept/manifest/wiki

[9] Linux Kernel Organization Inc. 2023. The Linux Kernel Archives. Retrieved July 31, 2023 from https://kernel.org/

[10] Microsoft Inc. 2023. Microsoft Windows. Retrieved July 31, 2023 from https://www.microsoft.com/en-us/windows

[11] Synopsys Inc. 2023. Coverity Static Application Security Testing. Retrieved July 31, 2023 from https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html

[12] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. http://xavierleroy.org/publi/compcert-CACM.pdf

[13] User "mtjm". 2012. Cross compilation fails using system linker for other architecture binaries. Retrieved July 31, 2023 from https://gitlab.haskell.org/ghc/ghc/-/issues/6086

[14] German Federal Ministry of Education and Research. 2021. VerSeCloud Project Description. Retrieved July 31, 2023 from https://www.forschung-it-sicherheit-kommunikationssysteme.de/projekte/versecloud

[15] Rajan Patel. 2022. An overview of live kernel patching. Retrieved July 31, 2023 from https://ubuntu.com/blog/an-overview-of-live-kernel-patching

[16] Eduardo Rafael. 2021. *ReasonML Mobile.* Retrieved July 31, 2023 from https://github.com/EduardoRFS/reason-mobile

[17] The AlgoProg Team. 2019. *OCaPIC.* Retrieved July 31, 2023 from http://www.algo-prog.info/ocapic/

[18] The Coq Team. 2023. The Coq Proof Assistant. Retrieved July 31, 2023 from https://coq.inria.fr/

[19] The Cppcheck team. 2023. CppCheck. Retrieved July 31, 2023 from https://cppcheck.sourceforge.io/

[20] The Glasgow Haskell Team. 2019. GHC cross compiling - Terminology and background. Retrieved July 31, 2023 from https://gitlab.haskell.org/ghc/ghc/-/wikis/building/cross-compiling#terminology-and-background

[21] The Glasgow Haskell Team. 2023. List of tools needed to build GHC. Retrieved July 31, 2023 from https://gitlab.haskell.org/ghc/ghc/-/wikis/building/preparation/tools#list-of-tools-needed-to-build-ghc

[22] The MirageOS Team. 2022. MirageOS. Retrieved July 31, 2023 from https://github.com/mirage

[23] The MirageOS Team. 2022. *ocaml-solo5.* Retrieved July 31, 2023 from https://github.com/mirage/ocaml-solo5

[24] The Solo5 Team. 2023. Solo5 - A sandboxed execution environment for unikernels. Retrieved July 31, 2023 from https://github.com/solo5/solo5

[25] INRIA The OCaml Team. 2022. Installing OCaml from sources on a Unix(-like) machine. Retrieved July 31, 2023 from https://github.com/ocaml/ocaml/blob/trunk/INSTALL.adoc#prerequisites

[26] INRIA The OCaml Team. 2022. *OCaml 4.14.1.* Retrieved July 31, 2023 from https://github.com/ocaml/ocaml/tree/4.14.1

[27] Sergei Trofimovich. 2020. Cross-compilation from linux to windows broke (stage1 uses local rts headers, not system ones). Retrieved July 31, 2023 from https://gitlab.haskell.org/ghc/ghc/-/issues/18143

[28] Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. 2018. A Generic Virtual Machine Approach for Programming Microcontrollers: the OMicroB Project. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018).* Toulouse, France. https://hal.sorbonne-universite.fr/hal-01705825

[29] Perry Wagle. 2011. *addnas_ocaml-cross-compiler.* Retrieved July 31, 2023 from https://github.com/wagle/addnas_ocaml-cross-compiler

[30] Catherine 'whitequark' and Romain Beauxis. 2023. *OCaml cross-toolchains and cross-packages.* Retrieved July 31, 2023 from https://github.com/ocaml-cross

A ONLINE RESOURCES

The OSL4 bundle is located at the GitLab instance[5] of the University of Potsdam, with is publicly accessible. The L4Re Operating System Framework is publicly accessible via the GitHub repository[8].

Received 31 July 2023; revised 01 August 2023; accepted 5 June 2009