Software Verification with ITPs Should Use Binary Code Extraction to Reduce the TCB
(short paper)

Ramana Kumar[1], Eric Mullen[2], Zachary Tatlock[2], and Magnus O. Myreen[3]

[1] Data61, CSIRO / UNSW, Australia
[2] University of Washington, USA
[3] Chalmers University of Technology, Sweden

Abstract. LCF-style provers emphasise that all results are secured by logical inference, and yet their current facilities for code extraction or code generation fall short of this high standard. This paper argues that extraction mechanisms with a small trusted computing base (TCB) ought to be used instead, pointing out that the recent CakeML and Œuf projects show that this is possible in HOL and within reach in Coq.

1 Introduction

Software verification is a primary use of interactive theorem provers (ITPs). To verify a system, one uses the logic of the prover to model the system, specify its desired properties, and prove that the model satisfies the specification. To run the verified system, many provers facilitate *extracting code* from the model, to be compiled and executed in a mainstream functional language. While widely used, this approach leads to an unsettlingly large *trusted computing base* (TCB) – the unverified components a system depends on for correct construction and execution. The TCB of conventional extraction includes pretty printers that "massage" code into the target language (e.g., OCaml or Haskell) without proof, as well as the target language implementation (unverified compiler and runtime).

Recently, two verification projects, CakeML and Œuf, have shown that code extraction with a small TCB is possible without a significant increase in verification effort. They enable ordinary software verification in a prover, and (mostly automated) compilation of a verified system *within the prover* to extract *binary code* for execution. The TCB of this approach no longer includes sophisticated compilation or extraction machinery; binary "extraction" simply prints the literal bytes of verified machine code to a file outside of the ITP.

In this paper, our first contribution is to explain the principles of the binary extraction approach taken by CakeML and Œuf to reduce the TCB. For CakeML, this is the first demonstration of the toolchain outside of compiler bootstrapping. Our second contribution is a detailed account of what remains in the TCB after applying binary code extraction, including sketches of how to build a validation story for the remaining assumptions. We argue that, given the feasibility of these ideas, all ITP-based software verification projects should strive for a small TCB.

2 Binary Code Extraction Workflow

The main idea of binary extraction is to *stay inside the prover* until you have binary code. We show how this workflow naturally extends the conventional approach with a simple example: computing the frequencies of words in a file.

First, we specify correctness by defining valid_wordfreq_output, a predicate that holds when *output* represents the word frequencies of *file_contents*:

$$\begin{array}{l}\mathsf{valid\_wordfreq\_output}\ \mathit{file\_contents}\ \mathit{output} \iff \\ \exists\, \mathit{ws}. \\ \quad \mathsf{set}\ \mathit{ws} = \mathsf{set}\ (\mathsf{words\_of}\ \mathit{file\_contents}) \wedge \mathsf{sorted}\ (\lambda\, x\ y.\ x < y)\ \mathit{ws} \wedge \\ \quad \mathit{output} = \mathsf{concat}\ (\mathsf{map}\ (\lambda\, w.\ \mathsf{format\_output}\ (w, \mathsf{frequency}\ \mathit{file\_contents}\ w))\ \mathit{ws})\end{array}$$

An *output* is correct if there is a list of words *ws* such that the set of words in *ws* is the same as the set of words in *file_contents*; each element in *ws* is strictly less than the next one; and *output* is a concatenation of lines, each of which corresponds to an element of *ws* and contains the frequency of that word in *file_contents* formatted according to a format_output function.

Next, we implement the specification as a functional program inside the logic. The main function for our example, compute_wordfreq_output, is defined below using helpers: insert_line inserts each word from a line into an ordered binary tree; to_list flattens an ordered binary tree into a sorted list; and map and foldl are the usual functions over lists. The format_output and words_of functions are the same as in the specification.

$$\begin{array}{l}\mathsf{compute\_wordfreq\_output}\ \mathit{input\_lines} = \\ \mathsf{map}\ \mathsf{format\_output}\ (\mathsf{to\_list}\ (\mathsf{foldl}\ \mathsf{insert\_line}\ \mathsf{empty\_tree}\ \mathit{input\_lines})) \\ \\ \mathsf{insert\_line}\ t\ \mathit{line} = \mathsf{foldl}\ \mathsf{insert\_word}\ t\ (\mathsf{words\_of}\ \mathit{line})\end{array}$$

After defining the functional implementation, we prove that it computes the desired result. This is where most of the manual proof effort associated with ITP-based verification is applied. However, by sticking with shallowly embedded logical functions, we avoid the explosion of details that would arise in a program logic over a deep embedding. For the word frequency example, we prove the following theorem in about 100 lines of tactic-based proof script.

$$\begin{array}{l}\vdash \mathsf{valid\_wordfreq\_output}\ \mathit{file\_contents} \\ \quad (\mathsf{concat}\ (\mathsf{compute\_wordfreq\_output}\ (\mathsf{lines\_of}\ \mathit{file\_contents})))\end{array} \tag{1}$$

Conventional approach. One would now use code extraction to pretty print the verified program into a mainstream functional language like Haskell or OCaml — crucially without any proof relating a formal semantics of the target language with either the ITP-generated code or the functions in the logic.

Binary extraction approach. The proposed *binary extraction* approach takes a different route, which stays in the logic longer and produces proved guarantees about the behaviour of the generated code. This route requires some infrastruc-

ture — a synthesis tool and a verified compiler — but these need only be built once. Here are the steps of binary extraction:

S1: *Use proof-producing or verified synthesis* to translate the shallowly embedded logical functions, such as compute_wordfreq_output, into a deep embedding in a programming language with a formal semantics. The result is a pure program that is proved to perform the desired computation.

S2: *Add some verified wrapper I/O code* so that the program from S1 can interact with its environment. The result is a complete standalone program. The previous step (S1) is automatic, but this step (S2) might be interactive or automatic depending on the desired I/O interaction.

S3: *Compile the verified standalone program in the logic* so that the compiler's correctness theorem can be applied to a theorem about its evaluation, i.e., ⊢compile source = compiler_output for a particular source and compiler_output.

These steps yield a theorem stating that compiler_output is a machine-code program that performs I/O according to S2 and implements the computation from S1, which will have been verified against its specification using typical ITP methods. For our example, the computation is compute_wordfreq_output, so we connect the behaviour of compiler_output to the valid_wordfreq_output property via the algorithm-level theorem (1). Compared with the conventional approach, the only extra manual effort in S1–S3 is the verification of the I/O wrapper in S2.

Below, we show how S1–S3 are realised in HOL4 by CakeML, and how they are almost realised in Coq by Œuf. That these ideas are supported (or very nearly supported) in both provers demonstrates cross-prover applicability.

Binary extraction in CakeML. S1–S3 are supported by different parts of the CakeML ecosystem and the underlying HOL4 theorem prover. (S1:) Functional programs written in HOL are translated to pure CakeML functions by an automatic proof-producing synthesis tool [21]; (S2:) the wrapper code is added to the code from S1 manually and verified using characteristic formulae (CF) for CakeML [8]; and (S3:) the verified CakeML compiler's backend [23] is evaluated in the logic using HOL4's evaluation engine by Barras [2]. S1 and S3 are automatic, while S2 currently requires some expertise from the user.

For our example, S2 involves writing a wrapper like the code shown below and proving a CF separation-logic-style correctness theorem for it.

```
val _ = (append_prog o process_topdecs) ‘
  fun wordfreq u =
    case TextIO.inputLinesFrom (List.hd (CommandLine.arguments()))
    of SOME lines =>
      TextIO.print_list (compute_wordfreq_output lines) ‘
```

The first line above instructs HOL4 to add code to the CakeML program being constructed. The code between the quotation marks, ‘ ... ‘, is CakeML concrete syntax for the top-level CakeML function. `compute_wordfreq_output` is

the synthesised CakeML function corresponding to the compute_wordfreq_output HOL function. Other names refer to functions in the CakeML basis library.

Once S1–S3 have been completed, the theorems from each step are easily composed to produce an end-to-end correctness theorem[4], which we explain below.

$$
\begin{array}{l}
\vdash \mathsf{wfCL}\ [\mathit{pname};\ \mathit{fname}] \wedge \mathsf{wfFS}\ \mathit{fs} \wedge \mathsf{hasFreeFD}\ \mathit{fs} \wedge \\
\quad \mathsf{get\_file\_contents}\ \mathit{fs}\ \mathit{fname} = \mathsf{Some}\ \mathit{file\_contents} \wedge \\
\quad \mathsf{x64\_installed}\ \mathsf{compiler\_output}\ (\mathsf{basis\_ffi}\ [\mathit{pname};\ \mathit{fname}]\ \mathit{fs})\ \mathit{mc}\ \mathit{ms} \Rightarrow \\
\quad \exists\, \mathit{io\_events}\ \mathit{ascii\_output}. \\
\qquad \mathsf{machine\_sem}\ \mathit{mc}\ (\mathsf{basis\_ffi}\ [\mathit{pname};\ \mathit{fname}]\ \mathit{fs})\ \mathit{ms} \subseteq \\
\qquad \mathsf{extend\_with\_resource\_limit}\ \{\ \mathsf{Terminate\ Success}\ \mathit{io\_events}\ \} \wedge \\
\qquad \mathsf{extract\_fs}\ \mathit{fs}\ \mathit{io\_events} = \mathsf{Some}\ (\mathsf{add\_stdout}\ \mathit{fs}\ \mathit{ascii\_output}) \wedge \\
\qquad \mathsf{valid\_wordfreq\_output}\ \mathit{file\_contents}\ \mathit{ascii\_output}
\end{array}
\tag{2}
$$

The first two lines make assumptions about the environment, namely: the command line must consist of two well-formed (wfCL) words, *pname* and *fname*; the file system *fs* must be well-formed (wfFS) with a free file descriptor (hasFreeFD); and *fname* must exist in *fs* with contents *file_contents*. The third line is more interesting and concerns the initial machine state *ms*. We assume that *ms* is an x86-64 machine state where compiler_output has been installed into memory and is ready to go; we also assume that CakeML's foreign-function interface (basis_ffi) behaves according to our model of the file system and standard streams.

If all these assumptions are true, then the machine-code level execution (machine_sem) will terminate. During execution the machine will perform some *io_events* (or some prefix of them, if it runs out of memory). The extract_fs line states that running the file system model *fs* through the *io_events* has the effect of adding some *ascii_output* to standard output. The last line states that this *ascii_output* is correct according to our specification valid_wordfreq_output.

Binary extraction in Œuf. Œuf accomplishes S1–S3 similarly to CakeML, but in Coq and building on CompCert [16]. (S1:) A Gallina program is translated to Œuf functions by an untrusted Coq plugin and translation validated to ensure equivalence [20]. Œuf then compiles the code to CompCert's Cminor IR. (S2:) I/O wrapper code is written in C, compiled to Cminor via CompCert, and linked to the code from S1. (S3:) The combined stand-alone program is compiled to assembly using CompCert, relying on tools like Valex to formally validate assembling [13]. An extracted version of the Œuf compiler is still currently used, since CompCert is not yet fully executable within Coq [17].

For our example, a user would write an I/O wrapper in C similar to:

```
int main(void) { union list* input = to_coq_str(read_stdin());
                 union list* freqs = OEUF_CALL(wordfreq, input);
                 write_stdout(of_coq_str(freqs)); return 0; }
```

[4] https://code.cakeml.org/tree/master/tutorial/solutions

The `OEUF_CALL` macro constructs a closure and passes arguments to Œuf-extracted code while `to_coq_str` and `of_coq_str` translate between the Œuf string representation (lists of Boolean 8-tuples) and the standard C representation (`char*`). Proving S2 requires showing that C data conversions and system calls adhere to the Œuf ABI [20], which we have specified for this example.[5] Assuming these specifications, composing theorems for S1–S3 yields an end-to-end guarantee:

$$\begin{array}{l}\mathsf{Oeuf.compile}(\mathsf{wordfreq}') = \mathsf{OK\ c}\ \wedge\ \mathsf{Oeuf.link}(\mathsf{c}, \mathsf{shim}) = \mathsf{OK\ p}\ \wedge \\ \mathsf{CompCert.compile}(\mathsf{p}) = \mathsf{OK\ b}\ \wedge\ \mathsf{initSt}(\mathsf{b}, s_1) \Rightarrow \\ \exists\ w\ \tau\ s_2.\ s_1 \xrightarrow{\tau} s_2\ \wedge\ \mathsf{finalSt}(\mathsf{b}, s_2)\ \wedge \mathsf{stdIn}(\tau) = w\ \wedge\ \mathsf{stdOut}(\tau) = \mathsf{wordfreq}(w)\end{array} \tag{3}$$

The first two lines relate the original Gallina function, shim (wrapper), and compiled output for the whole program; and require that state s_1 is a valid initial state, (i.e., that b has been correctly loaded). The final line guarantees that, under these assumptions, the program will safely execute and terminate in a final state[6] (as CompCert assembly semantics are deterministic) while generating trace τ of I/O events and that this trace corresponds to reading string w from standard input and writing wordfreq(w) to standard output. stdIn and stdOut filter the trace and relate low-level values to Gallina values.

3 Trusted Computing Base

The binary extraction approach yields both a proved result (theorem (2) or (3)), and the verified binary executable itself (compiler_output or b) printed into a file. To show what remains in the TCB for correct execution, we analyse the CakeML version of the word frequency example.

The ITP: its logic and implementation. We trust our theorem prover: that classical higher-order logic is consistent [22, 10, 14], and that the HOL4 kernel implements this logic correctly. Trusting the ITP implementation means trusting the $\sim$4000 lines of Standard ML code in the kernel, the rationale underlying HOL4's LCF-based design [19], and the compiler (Poly/ML), OS, and hardware on which HOL4 runs. It is possible (though we have not done it here) to obtain externally checkable proof certificates from HOL4 (via OpenTheory [12]), mitigating the need to trust any specific ITP implementation. These kinds of trust are intrinsic to any ITP-based approach.

The specification. We need to correctly formalise the desired behaviour of our program (valid_wordfreq_output), because it is not checked by any proofs. However, a specification can be tested by proving sanity-checking theorems about it and evaluating (the executable parts of) its definition on concrete examples. Trust across this *specification gap* is intrinsic to any kind of formal verification.

The extraction procedure. To execute verified code, it must at some point exit the theorem prover and appear in memory associated with a running process. We trust the function — a very simple one for binary extraction —

[5] `https://github.com/uwplse/oeuf/tree/master/demos/word_freq`

[6] No resource limits are assumed since CompCert semantics model infinite memory.

that reads the code (as a term in logic) and prints it into an executable image template. We trust that this file is not tampered with, and that the linker (next paragraph) and OS loader operate correctly. These assumptions are captured in the x64_installed predicate, which specifies the expected state of the machine after the executable is loaded. In addition to carefully defining x64_installed, we could validate this assumption using runtime checks on startup: e.g., that the registers pointing to the ends of the CakeML heap are valid and aligned. Trusting something between formal models and reality is unavoidable.

The execution environment. The final theorem is about execution of a formal machine model (machine_sem). We trust the hardware to behave according to this model, and that the OS and other processes do not interfere with the CakeML process. (We model interference and assume it avoids the CakeML process's memory [7].) Machine models, like Fox's L3 models that we use, can be validated by systematic testing against the hardware [4, 6]. Our verified program interacts with its environment, and we model how we expect the environment to behave, with functions like basis_ffi and extract_fs. The I/O facilities (command line, files, and standard streams) available in CakeML's basis library are supported by a small C interface to the underlying system calls (e.g., `open`). We trust our implementation of this interface, and the C compiler (on the interface code only) and linker. The verified program may exit prematurely if it runs out of memory: we eliminate this occurrence only by observation.

4 Broader Context and Vision

The ITP community is pursuing several approaches relevant to reducing the TCB of code extraction. Important aspects of extraction have been proven correct for Coq [18] and Isabelle/HOL [3, 9]; the CertiCoq [1] team and Hupel & Nipkow [11] are working toward verified code generators for Coq and Isabelle/HOL respectively; and frameworks like the Isabelle Refinement Framework [15] and Fiat [5] are exploring other approaches to proof-producing code extraction. CakeML and Œuf are distinguished by striving to be a natural replacement for conventional extraction, using conventional programming languages for synthesis, and aiming to completely eliminate the compiler from the TCB by proving results about the behaviour of the whole program binary including effectful wrapper code.

Given the advances from throughout the community and the fact that similar results are supported across different ITPs, we feel that extraction with a small TCB is on the cusp of wide-scale feasibility for verified systems. Much is left to study and build before these approaches achieve the convenience and performance of conventional extraction techniques, but we have demonstrated that it is already possible to rigorously connect facts established in the logic of an ITP to binary executable code under a substantially smaller TCB, and without substantial increase in verification effort. We enthusiastically urge the rest of the ITP community to adopt and advance the ideas behind binary code extraction.

References

1. Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O.S., Sozeau, M., Weaver, M.: CertiCoq: A verified compiler for Coq. In: CoqPL (2017)
2. Barras, B.: Programming and computing in HOL. In: TPHOLs (2000)
3. Berghofer, S., Nipkow, T.: Executing higher order logic. In: TYPES (2002)
4. Campbell, B., Stark, I.: Randomised testing of a microprocessor model using SMT-solver state generation. SCP 118, 60–76 (2016)
5. Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. In: POPL. pp. 689–700 (2015)
6. Fox, A.C.J., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: ITP. pp. 243–258 (2010)
7. Fox, A.C.J., Myreen, M.O., Tan, Y.K., Kumar, R.: Verified compilation of CakeML to multiple machine-code targets. In: CPP. pp. 125–137 (2017)
8. Guéneau, A., Myreen, M.O., Kumar, R., Norrish, M.: Verified characteristic formulae for CakeML. In: ESOP. pp. 584–610 (2017)
9. Haftmann, F., Nipkow, T.: A code generator framework for Isabelle/HOL. In: TPHOLs (2007)
10. Harrison, J.: Towards self-verification of HOL light. In: IJCAR. pp. 177–191 (2006)
11. Hupel, L., Nipkow, T.: A verified compiler from Isabelle/HOL to CakeML. In: Ahmed, A. (ed.) European Symposium on Programming (ESOP). Springer (2018)
12. Hurd, J.: The OpenTheory standard theory library. In: NFM. pp. 177–191 (2011)
13. Kästner, D., Leroy, X., Blazy, S., Schommer, B., Schmidt, M., Ferdinand, C.: Closing the Gap – The Formally Verified Optimizing Compiler CompCert. In: SSS'17: Safety-critical Systems Symposium 2017. pp. 163–180. Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium, CreateSpace, Bristol, United Kingdom (Feb 2017), `https://hal.inria.fr/hal-01399482`
14. Kumar, R., Arthan, R., Myreen, M.O., Owens, S.: Self-formalisation of higher-order logic - semantics, soundness, and a verified implementation. JAR 56(3), 221–259 (2016)
15. Lammich, P.: Refinement to Imperative/HOL. In: ITP (2015)
16. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd ACM symposium on Principles of Programming Languages. pp. 42–54. ACM Press (2006)
17. Leroy, X.: Using coq's evaluation mechanisms in anger. `http://gallium.inria.fr/blog/coq-eval/` (2015)
18. Letouzey, P.: Extraction in Coq: An overview. In: CiE (2008)
19. Milner, R.: LCF: A way of doing proofs with a machine. In: MFCS. pp. 146–159 (1979)
20. Mullen, E., Pernsteiner, S., Wilcox, J.R., Tatlock, Z., Grossman, D.: Œuf: Minimizing the Coq extraction TCB. In: CPP '18. pp. 172–185
21. Myreen, M.O., Owens, S.: Proof-producing translation of higher-order logic into pure and stateful ML. JFP 24(2-3), 284–315 (2014)
22. Pitts, A.M.: The HOL System: Logic, 3rd edn., `https://hol-theorem-prover.org#doc`
23. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A.C.J., Owens, S., Norrish, M.: A new verified compiler backend for CakeML. In: ICFP. pp. 60–73 (2016)