Translation Validation for a Verified OS Kernel

Thomas Sewell
NICTA & UNSW, Sydney, Australia
thomas.sewell@nicta.com.au

Magnus Myreen
Cambridge University, UK
magnus.myreen@cl.cam.ac.uk

Gerwin Klein
NICTA & UNSW, Sydney, Australia
gerwin.klein@nicta.com.au

Abstract

We extend the existing formal verification of the seL4 operating system microkernel from 9 500 lines of C source code to the binary level. We handle all functions that were part of the previous verification. Like the original verification, we currently omit the assembly routines and volatile accesses used to control system hardware.

More generally, we present an approach for proving refinement between the formal semantics of a program on the C source level and its formal semantics on the binary level, thus checking the validity of compilation, including some optimisations, and linking, and extending static properties proved of the source code to the executable. We make use of recent improvements in SMT solvers to almost fully automate this process.

We handle binaries generated by unmodified gcc 4.5.1 at optimisation level 1, and can handle most of seL4 even at optimisation level 2.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability—Verification; D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Verification, Languages

Keywords Binary Verification, seL4, Microkernel

1. Introduction

Our aim is to extend one of the largest formal verifications, that of the seL4 microkernel by Klein et al [11], from around 9 500 lines of C source code down to 11 736 instructions on the binary level.

Recent successes in the formal verification of sizeable programs at the implementation level, such as the CompCert C compiler [14], the Verisoft project [1], or the above mentioned seL4 microkernel [11], suggest that formal verification of small, high-importance software systems is feasible and may become more common in the future. Unfortunately such software systems tend to be written in old, low-level, but high-performance languages like C whose standards are difficult to formalise, are by necessity purposely violated by systems programmers, and are rarely implemented precisely by their toolchains. This leaves the prospective verifier with a difficult choice. Clean-room designs of modern low-level systems languages are an attractive research topic, but tend to come with pragmatic limitations, such as requiring garbage collection for type safety. The alternative is to proceed with the best available semantic model of the implementation language and program and hope that any compiler defect or difference between the compiler's and the verifier's interpretation can be found by testing or other means.

In their study on compiler testing, Yang et al reported 325 previously unknown defects in 11 different C compilers [34]. Even the formally verified CompCert was found to exhibit 5 defects, albeit in its unverified front-end only—it was the only compiler Yang et al did not manage to break in its code generation phase, despite devoting significant resources to the task. We have used this verified compiler on the verified C source of the seL4 microkernel, but in this case found the result unsatisfactory: There is a remaining chance of a mismatch between CompCert's interpretation of the C standard in the theorem prover Coq and the interpretation of the C standard the seL4 verification uses [32] in the theorem prover Isabelle/HOL [21], esp. in cases where the standard is purposely violated to implement machine-dependent, low-level operating system (OS) functionality. Reconciling these two semantics is non-trivial. Firstly the logics of Coq and Isabelle/HOL are not directly compatible. Secondly, since the seL4 semantics is largely shallowly embedded, an equivalence proof would have to be performed for each program, similar to our more direct method below.

In this paper we present a hybrid approach, in which we take precisely the semantics for that specific program (seL4) produced by Norrish's C parser [32] for Isabelle/HOL as used in the seL4 verification. We then produce a model of the compiled binary of this program, extending the approach of Myreen [17, 18] in the HOL4 theorem prover [29], and we check the soundness of this extracted model against the strongly validated Cambridge semantics for the ARM architecture [8]. Finally we use an SMT-based proof process to establish that the compiled binary matches the expected semantics of the C source. Formally, we prove that the binary is a refinement of the stated C source semantics. This property composes with the existing functional correctness proof of seL4.

Our method eliminates one of the Achilles' heels of the verified compiler approach: the parser and lexer for concrete C syntax. Since the high-level functional verification and the binary verification connect to precisely the same formal artefact, the output of the now untrusted C parser, the method by which we arrive at this artefact is irrelevant. Instead, this trust is replaced by trust in the binary verification tool and into the import of an executable file into hexadecimal numbers into the theorem prover HOL4.

In fact, it is not even important any more that the compiler conforms to the C standard, or that the C semantics in the theorem prover agrees with the C standard. All that matters is that compiler and prover semantics match and thereby transport high-level properties that have been proved on the C code down to the binary level. For seL4, this includes functional correctness, integrity, authority confinement, and non-interference [11, 15, 28].

We provide a full proof of correctness of the compilation by standard gcc 4.5.1 of all previously formally verified functions [11] in seL4 at optimisation level 1. We additionally produce proofs for all of seL4's unverified initialisation functions except for 3

PLDI '13 June 2013, Seattle, WA.

nested loops. At optimisation level 2, which is the standard high-performance setting for seL4, we are able to automatically extract a model for all but four assembly routines (i.e. coverage of 98 %) and produce proofs of correctness for the vast majority of loop-free functions. Using optimisation level 1 rather than 2 incurs a performance cost of 15-20% depending on the system call.

Like the original seL4 verification, we do not handle the assembly routines and volatile accesses used to directly control the system hardware.

The verification strategy is to first build a representation of both the C program and the binary within a single restricted intermediate language. This language consists of simple control flow mechanisms and standard arithmetic operations which are common to the C language, CPU instruction sets and the SMT bit-vector theory. It is thus highly amenable to analysis by SMT solvers. To prove our refinement statement, we extract SMT formulas with consideration towards program loops as well as proof obligations and assumptions arising from the decompilation process and C semantics.

While our main target was extending the seL4 verification to the binary level, we believe that this approach generalises to other C verifications and to other compilers. We have for instance started experimenting with extending the decompilation phase to the output of CompCert. While there are still a number of limitations to overcome that we discuss in Section 4, we believe this work represents evidence for a general, flexible method of extending formal reasoning about C programs to the binary level using standard off-the-shelf compilers.

In the remainder of this paper, we first summarise the pieces of previous work we build on in Section 2, before describing the conversions and proof process in Section 3. Section 4 evaluates the performance and limitations of our verification method in more detail. We discuss related binary verification approaches in Section 5.

2. Background

In this section, we describe previous background work our binary verification of seL4 builds on. In particular, these are existing proofs on seL4 [11] together with its C semantics [32], and existing work on decompilation [17] and its validated ARM semantics [8].

2.1 The seL4 microkernel verification

The seL4 kernel is a general-purpose OS microkernel measuring about 9 500 source lines of C code[1] and 600 lines of assembly code. As usual for microkernels, it provides a minimal set of mechanisms, including threads, interrupts, virtual memory, and inter-process communication. Less usual is its capability-based mandatory access control model and explicitly user-controlled, but kernel-enforced memory allocation mechanism. It can be configured as a classical microkernel, as a hypervisor running Linux instances, or as a high-assurance pure separation kernel.

What makes this kernel special is that its functional correctness has been verified over multiple levels by formal refinement from an abstract specification down to the C source code level [11], and that further properties have been established over that specification [28]. It represents a large body of existing work, with over 200 000 lines of Isabelle/HOL proof, that our approach has to interface to if it is to extend this verification to the binary level.

Figure 1 illustrates the existing verification stack on top and our new work interfacing with it on the bottom. The existing work starts with high-level properties on top, abstract functional specification next, design-level specification further down, and the semantic C source code level at the bottom. Our work extends this stack by the semantic description of the binary, thereby removing compiler and Isabelle/HOL C parser from the trust chain of the proof and replacing it with the ARM semantics and our tool set.

[1] Klein et al report 8 700 lines [11]. We are basing our work on a newer version of seL4 that supports further API features.

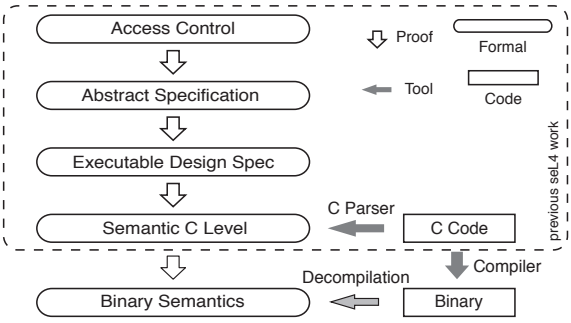

Figure 1. Existing seL4 verification stack.

The kernel exists for multiple architectures, among them Intel x86 (unverified) and ARMv6/ARMv7 (verified). It is available as a commercial product under the name OKL4:verified and for free download for academic use [22].

Apart from its manageable size, key seL4 features for verifiability were that, as usual for microkernels, almost all device drivers and therefore most hardware interaction is outside the kernel in user-space, and the fact that seL4 is an event-based kernel for a uni-processor setting. This means, interrupts are switched off for most kernel operations. For long-running operations, seL4 contains specific preemption points where it explicitly checks for interrupts. This allows the verification to proceed on a sequential language model where the explicit coarse-grained preemption points are normal kernel exits. This is important, both for the C semantics used in the original verification as well as for the ARM semantics used in the binary decompilation work we build on.

As illustrated in Figure 1, the bottom level of the existing verification work is a semantic Isabelle/HOL model of the C source code of seL4, generated automatically by Norrish's C parser [32]. This parser instantiates a generic language framework for imperative languages by Schirmer [27], consisting of a generic operational semantics framework with a Hoare-logic and verification condition generator on top, which are proved sound and relatively complete in Isabelle/HOL. This instantiation covers a large subset of standard C99 [10] extended with architecture and compiler-specific assumptions, for instance about endianness and data layout, that systems programmers have to make. Its largest limitation is that it does not permit taking the address of local variables. This enables the C model to treat parameters and local variables on the stack separately from the pointer-addressable heap, thereby simplifying verification. This limitation could be lifted in future work at the expense of having to invest more explicit reasoning about this separation in C source-level verifications.

The memory model this parser uses in its semantic C embedding is due to Tuch [31, 32]. Its main feature is that it provides abstract C types and comparatively convenient type-based reasoning on top of a precise, byte-wise memory model that formalises the C heap merely as a function from 32-bit words to 8-bit words. Type information is kept as a so-called *heap type description* in ghost state for verification convenience, and Isabelle/HOL type classes establish abstractions on top. At the rare occasions where kernel code breaks type safety, such as memory allocation, reasoning can fall back to the explicit byte level. This model enables high-level reasoning in the existing verification, and is also compatible with the binary level in this work.

2.2 Decompilation from ARM into Higher-Order Logic

In this paper, we present how we proved seL4 correct down to the concrete ARM machine code that gcc produces. In order to reason formally about ARM machine code we require a semantics for this machine code, i.e. a formal specification of the ARM instruction set architecture (ISA). For this purpose, we build on the Cambridge ARM ISA specification [8] and proof tools, namely a proof-producing decompiler, which makes reasoning about such complex ISA specifications tractable.

The Cambridge ARM specification has evolved from a string of ARM related projects that started in 2000 with a hardware verification project where Fox formally specified the ARM ISA version 3 and proved, using the HOL4 theorem prover [29], that a hardware implementation (the ARM6) implements its ISA correctly. Later, the project's focus shifted to software verification: the ARM model was updated and extended to cover all modern versions of the ARM ISA, namely versions 4–7 including all operation modes and every instruction. Since the latest model is no longer directly connected via proof to modern hardware implementations, Fox and Myreen have extensively validated the latest specification against different real hardware implementations [8]. This latest ARM specification is the most comprehensive formal specification of a commercial ISA that is publicly available.

Due to its history, the Cambridge ARM specification is highly trustworthy, lengthy and very detailed. Reasoning manually in a theorem prover about such models is hopelessly tedious. For this reason, Myreen et al. developed automation that makes machine-code verification tractable even for very complex ISA specifications. The main tool is called a *decompiler*. This tool aids program verification by extracting a piece of the lengthy model, given a program to interpret the model on. More specifically, given a snippet of machine code, this decompiler extract a function describing the effect of running the code on the ISA specification. Loops in the machine code turn into recursion in the extracted functions. The decompiler is automatic and proof producing: for each run, the decompiler proves that the extracted function is indeed accurate w.r.t. the given code and the ISA specification. Myreen [16] has shown that decompilation can be used for verification of sizeable case studies, e.g. garbage collectors and Lisp implementations. This paper builds on decompilation, extends it and shows that it can be used in the verification of the seL4 microkernel.

3. Correctness Proof

This section describes our refinement approach between C source semantics and binary code.

The verification process involves a collection of representations of the input program which are outlined in Figure 2. The two inputs into the process are the C program and binary ELF file on the top of Figure 2. The three dotted boxes in the diagram represent the three main proof systems used in the binary verification: the interactive LCF-style provers Isabelle/HOL and HOL4, and our SMT-based proof tool that is centred around a common intermediate language describing control-flow graphs. On the left of the diagram, the C program is parsed into the theorem prover Isabelle/HOL using existing tools [32], and then transformed into a form closer to binary code. The first such transformation step is within Isabelle/HOL, the second in our external tool. On the right of Figure 2, the binary program is decompiled into HOL4, using the existing validated Cambridge ARM semantics [8]. Since the logics of the two theorem provers are almost identical, these can be translated into Isabelle/HOL in a straightforward way to be exported to the intermediate graph language. Once the C program and binary have both been represented in the common graph language, on the bottom of

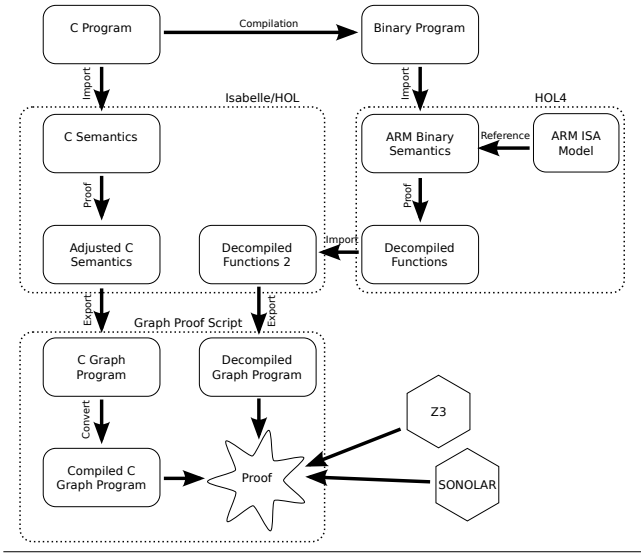

Figure 2. Artefacts in the correctness proof

the diagram, refinement between them can be proven with the assistance of the two SMT solvers Z3 [7] and SONOLAR [23].

The following subsections present each of these steps in more detail.

The SMT-based proof tool and some of the relevant Isabelle theories are available from `http://www.ssrg.nicta.com.au/software/TS/graph-refine/`.

3.1 Conversion from C Semantics to the Graph Language

This subsection describes the pseudo-compilation process which converts functions in the C model into the intermediate graph language. This also serves as an introduction to the graph language.

The idea of the graph conversion is to replace the language based control flow rules with a simpler control flow graph. An example conversion is given in Figure 3. All statements are numbered, and the steps between them become graph edges, giving us a labelled, directed graph. The point of doing this is that the context-dependent effects of `break`, `continue` and others are replaced by graph edges which simply specify the number of the next statement. The special label `Ret` represents return from the function.

The graph consists of three types of nodes. Conditional nodes are used to pick between execution paths, and correspond closely to decisions made by `if` and `while` statements in C. Basic nodes represent normal statements, and update the value of some variable with the result of some calculation (memory is represented as a variable). Call nodes are used to represent function calls, which are distinguished from other statements. A number of restrictions enforced by the C parser are relevant here: function calls may be embedded in other expressions and statements only in very limited ways, statements with multiple effects are forbidden, and switch statements must always be convertible into a chain of if-else statements.

The conversion also pseudo-compiles all C expressions. An example is on the right hand side of Figure 3. Insofar as C can be seen as a portable assembler, the conversion makes this explicit. Pointers become 32-bit words, with address operations of various kinds becoming explicit arithmetic. Local variables of structure and array type are replaced by collections of local variables representing their fields. Assignments of structure type are expanded into a sequence of assignments for each field. Reads and writes of global variables become reads and writes of memory, at symbolic addresses which

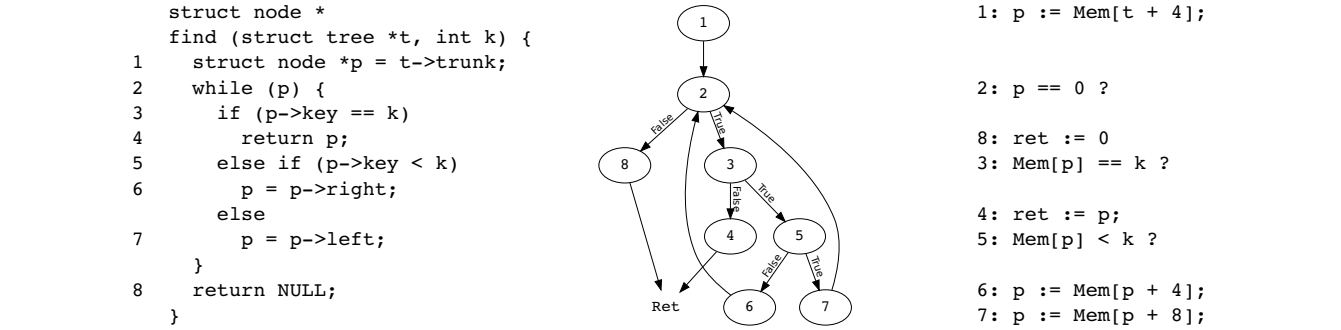

Figure 3. Example Conversion of Structure and Statements to Graph Language

are later instantiated by reading the ELF symbol table. The expressions that remain are entirely machine compatible: operations such as 32-bit addition and multiplication, left and right shifts, signed and unsigned less-than, and finally memory access and update of 32-bit and 8-bit values.

It is a theorem of the Tuch memory semantics [31] that memory writes of aggregate types are equivalent to sequences of writes of their fields, assuming the aggregate type contains no padding. Padding creates a number of headaches for us, and so for the purposes of this work we have adjusted the seL4 source to ensure that all structures of interest are packed (see Section 4.3).

The complication is that C is not merely a portable assembler. The C standard mandates a number of restrictions on the way various operations may be used, and some of these restrictions go beyond the scope of machine operations. One simple restriction is that arithmetic on signed operands may not overflow, and another is that dereferenced pointers must be aligned and nonzero. To ensure the standard is followed, the parser inserts a number of `Guard` statements into its output, which allow execution to continue only if some condition is met. The conditions become verification obligations in the existing verification work.

These guard statements are translated into condition nodes with one of the outbound edges pointing to the special label `Err`. Given that the guard conditions have all been established from the invariants in previous work, it will become an input assumption to the current work that these paths to `Err` are never taken. These guards were omitted from Figure 3 — there ought to be condition nodes immediately before all memory-using nodes which check pointer validity.

The key restriction from the C standard that cannot be realised in machine operations is the strict-aliasing rule. This allows the compiler to assume that a given memory address is not in use with two different incompatible types. In systems code, programmers occasionally break this assumption (see also Section 4.4), but most code conforms to it. Since optimisations frequently make use of the rule, we need to make the information it conveys available at the points where it does hold. To do this, we strengthened the checks made in the C parser output against the heap type description, a global variable which tracks the expected type of memory. The graph conversion then includes the heap type description and checks in its output. The stronger checks are of the form pvalid $htd\ \tau\ p$ for some heap type description htd, C type τ and pointer value p. The expected non-aliasing conditions are theorems of the Tuch memory model [31], for instance this rule about ints and floats:

$$\frac{\text{pvalid } htd \text{ int } p \quad \text{pvalid } htd' \text{ float } p}{\{x \mid p \leq x < p + 4\} \cap \{x \mid p' \leq x < p' + 4\} = \{\}}$$

The semantics of the graph language are straightforward to formalise in Isabelle/HOL or HOL4. The node types are introduced as datatype constructors `Basic`, `Cond` and `Call`. A single step starting from a `Basic` node updates local variables, and starting from a `Cond` node decides between two possible successor labels. The `Call` nodes create a new stack frame, with a new graph and new local variables, and steps from the `Ret` and `Err` labels fold the current stack frame into the previous one. The semantics of execution are given by the transitive closure of this single-step relation. Given this formal semantics, we have proven, in Isabelle/HOL, that the converted functions in the graph language refine the original C semantics. The details of this formalisation and proof are elided here, and are largely technical.

3.2 Decompiling Compiler Output into Logic

The next piece of the puzzle is the right-hand side of Figure 2, i.e. how we take binaries the compilers (recent versions of standard unmodified gcc) produce, decompile these binaries into functions, which we, in turn, translate into the graph language described above. By *decompilation* we mean proof-producing extraction of functional programs from concrete binaries. Here we build on previous work on such decompilation [17, 18], but in this text we do not assume any prior knowledge of the previous work. Instead, we use a few examples below to demonstrate what decompilation provides and what we had to alter in Myreen's original decompilation approach to make the decompiler's output better compatible with the C semantics to which the left-hand side of the Figure 2 connects the decompiler's output.

3.2.1 Simple Decompilation Example

To get a flavour of what decompilation provides, consider the following simple C function for taking the average of two integers.

```
uint avg (uint i, uint j) {
  return (i + j) / 2;
}
```

When compiled with gcc, this C code is translated into an ARM binary (an ELF file). Applying relevant `objdump` tools to the binary, one can produce a text file showing the generated ARM machine code (on the left below) and ARM assembly code (in the centre).

```
<avg>:
e0810000     add r0, r1, r0    // add r1 to r0
e1a000a0     lsr r0, r0, #1    // shift r0 right
e12fff1e     bx  lr            // return
```

To decompile the generated machine code, one simply provides the hex codes on the left to the decompiler together with the signature of the C function. The decompiler extracts from the machine code a function in logic that describes the state update the machine code performs. The machine code above is converted into the following logic function. Note that here, r_0 and r_1 are 32-bit integers

and arithmetic $+$ is over 32-bit integers (overflow wraps around).

$$\begin{array}{rcl} \mathsf{avg}\ (r_0, r_1) & = & \mathsf{let}\ r_0 = r_1 + r_0\ \mathsf{in} \\ & & \mathsf{let}\ r_0 = r_0 >\!\!>\!\!> 1\ \mathsf{in} \\ & & r_0 \end{array}$$

The decompiler also automatically proves a *certificate* theorem: a theorem relating the extracted function avg with the original machine code. These certificate theorems are stated in terms of a total-correctness Hoare logic with triples $\{pre\}code\{post\}$ for machine code. They are defined and proved in terms of the underlying specification of the ARM instruction set architecture [8]. Precise details on their definition be found elsewhere [16]. Informally, the following Hoare triple can be read to say: if the program counter p points to the start of the machine code and r_0, r_1 and lr hold initial values of registers 0, 1 and 14 respectively, then execution will reach a state where the postcondition is true, i.e. a state where the value of register 0 is described by $\mathsf{avg}\ (r_0, r_1)$ and the program counter is set to the return address lr. Informally, read $*$ below as 'and', formally this is a separating conjunction from separation logic [25] but here used to separate between any machine code resources [16].

$$\begin{array}{l} \{\ \mathsf{R0}\ r_0 * \mathsf{R1}\ r_1 * \mathsf{R14}\ lr * \mathsf{PC}\ p\ \} \\ p : \texttt{e0810000 e1a000a0 e12fff1e} \\ \{\ \mathsf{R0}\ (\mathsf{avg}\ (r_0, r_1)) * \mathsf{R1}\ \_ * \mathsf{R14}\ \_ * \mathsf{PC}\ lr\ \} \end{array}$$

The benefit of decompilation is that the extracted functions provide a convenient abstraction of the machine code from which it is much more tractable to perform further proofs. Further proofs need only deal with the extracted function because the certificate theorem states that the behaviour of the extracted functions and the original machine code agree according to the specification of the instruction set architecture.

How is this form of proof-producing decompilation implemented in a theorem prover? The original approach does not use any heuristics and blindly follows the following steps:

1. For each instruction in the machine code, evaluate the specification of the instruction set architecture and prove a machine-code Hoare triple describing the effect of each instruction.
2. Construct a control-flow graph (CFG) based on these Hoare triples. Split the code into separate decompilation rounds, e.g. one round for each (nested) loop.
3. For each decompilation round:
   (a) Compose the Hoare triples for each path through the code segment, merge these Hoare triples to produce a single Hoare triple describing this code segment.
   (b) If there is a jump to one of the entry points, then apply a special loop rule which introduces a tail-recursive function,
   (c) The result is a theorem stated as a Hoare triple. Its postcondition mentions the effect of executing the code segment, expressed as a function applied to the initial values. We read off this function from the postcondition and return both the (certificate) theorem and the extracted function.

This form of decompilation is described in detail elsewhere [16] and has recently been extended [18] to also allow arbitrary use of code pointers, which the original approach struggled to handle.

3.2.2 C-Compatible (Stack Aware) Decompilation

Decompilation as outlined above can effectively extract functions from machine code. However, the functions the decompilation produces are not immediately compatible with the C semantics which we target. An extension of the avg example from above illustrates what goes wrong. Consider the following C function which takes eight integers as input and calculates their average.

```
uint avg8 (uint i1, i2, i3, i4, i5, i6, i7, i8) {
  return (i1+i2+i3+i4+i5+i6+i7+i8) / 2;
}
```

When this is compiled, gcc produces the following ARM assembly. Note that arguments `i5-i8` are passed on the stack. Hence the memory load instructions `ldr` and `ldm`.

```
<avg8>:
e0811000     add r1, r1, r0
e0811002     add r1, r1, r2
e59d2000     ldr r2, [sp]        // load
e0811003     add r1, r1, r3
e0810002     add r0, r1, r2
e99d000c     ldmib sp, {r2, r3}  // load
e0800002     add r0, r0, r2
e0800003     add r0, r0, r3
e59d300c     ldr r3, [sp, #12]   // load
e0800003     add r0, r0, r3
e1a001a0     lsr r0, r0, #3
e12fff1e     bx   lr
```

The original decompiler knew nothing about how C uses the stack and thus treats the stack accesses as ordinary memory accesses. This results in extracted functions where stack accesses touch memory explicitly, e.g. the last load from above turns into a line:

$$\ldots\ \mathsf{let}\ r_3 = m(r_{13} + 12)\ \mathsf{in}\ \ldots$$

This fits very badly with the C semantics for which this spilling into the stack is completely hidden. The C semantics we use treats all local variables as simple 'register' variables, i.e. according to the C semantics `avg8` does not make any memory accesses.

To remedy this mismatch, we made the decompiler aware of the stack and the C calling convention for ARM. We made the decompiler treat the stack as a separate datastructure. We formalised a new separation-logic inspired stack assertion

$$\mathsf{stack}\ sp\ n\ stack$$

which is true if the stack pointer (register 13) holds a value sp such that the list of elements $stack$ are on the stack and there is space for n elements above the stack pointer, i.e. n elements can safely be pushed onto the stack. We can keep this datastructure separate from the rest of memory by proving pre/postconditions where the 'heap' memory is separate from the stack using the separating conjunction $*$ from separation logic [16, 25].

$$\mathsf{stack}\ sp\ n\ stack\ *\ \mathsf{memory}\ m$$

The altered decompiler now has a new *stack simulation* phase which attempts to discover where and how the stack pointer can travel through the code, i.e. it attempts to identify which instructions access the stack and what offsets are used, i.e. which stack elements are accessed. This phase comes immediately after CFG exploration (i.e. step 2 of the original algorithm).

When run on the simple `avg8` example, this stack heuristic finds that all the load instructions are stack accesses. For each of the stack accesses, it derives a new Hoare triple stated in terms of the stack assertion. For example, the last load in the example above is described by the following Hoare triple. Here :: is list cons.

$$\begin{array}{l} \{\ \mathsf{R3}\ r_3 * \mathsf{stack}\ sp\ n\ (s_0{::}s_1{::}s_2{::}s_3{::}ss) * \mathsf{PC}\ p\ \} \\ p : \texttt{e59d300c} \\ \{\ \mathsf{R3}\ s_3 * \mathsf{stack}\ sp\ n\ (s_0{::}s_1{::}s_2{::}s_3{::}ss) * \mathsf{PC}\ (p{+}4)\ \} \end{array}$$

Given these Hoare triples stated in terms of stack, the rest of the decompiler runs just as before. The Hoare triple above turns into a let-expression of the following form:

$$\ldots\ \mathsf{let}\ r_3 = s_2\ \mathsf{in}\ \ldots$$

With the stack-aware decompiling, the entire code for `avg8` turns into the following function which does not mention memory. We have expanded the let-expressions for brevity below.

$$\begin{array}{l}\mathsf{avg8}\ (r_0, r_1, r_2, r_3, s_0, s_1, s_2, s_3) = \\ \quad (r_1 + r_0 + r_2 + r_3 + s_0 + s_1 + s_2 + s_3) \ggg 3\end{array}$$

The generated certificate theorem is stated in terms of the stack assertion. For `avg8`, this theorem is:

$$\begin{array}{l}\{\mathsf{R0}\ r_0 * \mathsf{R1}\ r_1 * \ldots * \mathsf{stack}\ sp\ n\ (s_0{::}s_1{::}s_2{::}s_3{::}ss) * \mathsf{PC}\ p\} \\ \ p : \texttt{e0811000 e0811002 ... e12fff1e} \\ \{\mathsf{let}\ r_0 = \mathsf{avg8}\ (r_0, r_1, r_2, r_3, s_0, s_1, s_2, s_3)\ \mathsf{in} \\ \quad \mathsf{R0}\ r_0 * \mathsf{R1}\ \_ * \ldots * \mathsf{stack}\ sp\ n\ (s_0{::}s_1{::}s_2{::}s_3{::}ss) * \mathsf{PC}\ lr\}\end{array}$$

Care is taken to make these Hoare triples adhere exactly to the calling convention so that they can be used in future decompilations of code that call this function. Using this certificate theorem, we can decompile a call to `avg8` into a let-expression of the following form in the decompilation of a caller.

$$\ldots\ \ \mathsf{let}\ r_0 = \mathsf{avg8}\ (r_0, r_1, r_2, r_3, s_0, s_1, s_2, s_3)\ \mathsf{in}\ \ \ldots$$

Given that we do not allow C code to take the address of a local variable, this stack simulation is relatively straightforward to implement in practice, i.e. it is reasonably easy to automatically find which locations access the stack and which do not (even without use of any debug information). If we were to allow taking the address of a local variable, then this approach would fall apart very quickly as the distinction between memory and stack locations becomes blurred.

Having said that, there is one situation where the C compiler will produce code that passes an address of a stack location around, even if the C code does not seem to do so. This happens in cases where a function is to return a struct that does not fit into a single machine word, i.e. does not fit into a single register. When a function is to return a struct, it expects to get an address into the caller's stack space. This address points to a segment in the caller's part of the stack into which the callee is to write its result.

Our approach to dealing with the stack can manage this form of passing around of pointers into the stack. The solution is to initialise the stack heuristic appropriately. Instead of starting the stack heuristic off from a state where the caller's stack is an opaque untouchable part of state, we start it off with a slot for the return struct. If the result is n-words long, then we start the stack simulation from a stack which consists of an initial segment ss, the result slots $[t_1, t_2, \ldots, t_n]$ and finally the rest of the stack $stack$:

$$\mathsf{stack}\ sp\ n\ (ss +\!\!+ [t_1, t_2, \ldots, t_n] +\!\!+ stack)$$

We also assume (according to the calling convention) that register 0 holds, on entry to the function, the address of the first result slot in the stack, i.e. $sp + 4 \times \mathsf{length}\ ss$. With such an initialisation, we can deal with the case of returning a struct directly into the stack. The rest of the decompiler runs exactly as before.

The new stack heuristic brings with it a number of new limitations, since the decompiler now tries to discover what the compiler did with the stack and prove that the stack is kept separate from the heap. Limitations are discussed in Section 4.2.

3.2.3 Converting Extracted Functions into Graph Format

The pure functions extracted here can be easily converted into the graph representation. An example is shown in Figure 4. We invent a collection of unique variable names for the input values (x and y in the example), the (anonymous) output values (we pick $r0$ and $r1$ in the example), and each variable fixed in a let expression (a, b and c). If-then-else expressions become condition nodes. Values, such as $(3, x)$, become updates to variables, in this case to a and b as set by the let expression they appear in. Let expressions control order of operations but create no nodes. Function calls become function call nodes. Tail recursion becomes a graph arc back to the beginning of the function.

```
f x y =
  let (a, b) = if x < y
    then (1, 2)
    else (3, x)
  in let c = a + b - y
  in (c, 0)
```

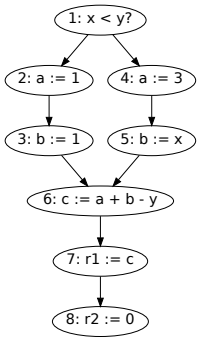

Figure 4. Example conversion of let-expression to graph

This conversion is currently done automatically, but without proof of correctness. If more assurance is desired, the transformation could in future work be included on the Isabelle/HOL side of the tool and produce a corresponding certificate theorem.

3.3 Proof of Refinement

The final step in our process is the refinement proof, which we decompose function by function. We assume that we can treat other called functions as black boxes fully specified by their corresponding refinement theorem, with the calling convention giving us enough information to relate the C and binary behaviour. If the compiler makes use of inter-procedural analysis this may invalidate our assumption; we currently do not support this case. This compositional approach reduces our large code base to a series of manageable problems.

As recommended in Pnueli et al's original definition of translation validation [24], we divide the validation process into a search process and a checker, with the search process discovering a proof script which the checker then validates. This separation is partial, with the two processes sharing some code and the proof steps checked being far from minimal. The proof script consists of a problem space description together with a tree of proof rules `Restrict`, `Split` and `Leaf`. The rules give structure to the proof, but all the heavy lifting is done by converting proof goals on the problem space into SMT problems. We will describe these proof components as the proof checker sees them.

3.3.1 Inlining and The Problem Space

The first step in building a proof script is to establish a problem space, a shared graph namespace into which the binary and C bodies of the function of interest are copied. The problem space is free to be modified, in particular by inlining function calls. The search process attempts to inline sufficiently that the two function graphs in the problem space can be proven equivalent.

Inlining is done on the binary side of the problem whenever the called function does not match any C function, which may be because the compiler invented the function or modified its signature.

Inlining is done on the C side of the problem at any call site which is reachable and which calls a function name that does not appear on the binary side. This simple heuristic may fail in the presence of selective inlining. It could also potentially inline far too much source code in the case where a function designated pure or const was dropped because its result was ignored. This heuristic has, however, worked well at the optimisation levels we currently address.

This heuristic considers the C and binary functions completely separately, ignoring the relationship between them for now. This is a deliberate design decision allowing us to perform all inlining before any other analyses.

The problem space produced after inlining is included in the proof script, and the checker trusts that this problem space is derivable from the functions of interest. Checking that inlining was performed correctly did not seem worthwhile.

3.3.2 Conversion to SMT

The search and checker processes both use SMT solvers extensively to make judgements about the C and binary execution. These executions will form a sequence of visits to nodes in the problem space graph. The items of interest for a given node n will be the values of variables, should n be visited, at the point in execution that n is reached, and also the conditions under which n is reached. The final objective of the proof process is to reason about the values of returned variables should `Ret` be reached, and the conditions under which `Err` is reached.

These valuations and conditions can be represented in the SMT logic. Figure 5 shows nearly all of the steps of interest. The boxes show the path condition and variable values immediately before execution of each node. The function input variables are simply named, for instance x_i, and are unknowns in SMT. The path condition at the starting node, 1, is simply true. Basic nodes, such as 1 and 3, update the variable state with values taken from the existing variable state substituted into their expression, for instance, at node 1, $x+1$ is evaluated as x_i+1, and this is used to update the value of y. Condition nodes, such as 2, substitute their expression and add it to the path condition, such as the path condition at 3. When paths converge, such as at 4, the path condition to the node is the union of the conditions on each path, and the variable values are constructed using if-then-else expressions — if the path via 3 was taken, x has the value from 3, otherwise from 2.

Conveniently omitted from Figure 5 was the variable state after calling function f at 4. The return value is named, for instance z_after_4, and becomes another SMT unknown. The SMT conversion process notes the input and output values, and if any other call to f is made, adds SMT assertions that f will return the same values given the same inputs. If the two calls are on the C and binary side, the argument types may differ slightly, in which case the assertion takes into account the calling convention.

Figure 5 is inaccurate in that it is fully expanded. The expressions computed at nodes 1 and 2 would have been given names using SMTLIB2's definition feature, for instance $y\_after\_1 = x_i+1$, $cond\_at\_2 = y\_after\_1 < 3$. This reduces the syntactic expansion that already is seen at node 4.

This process does not handle loops, and the generalisation will be discussed in Section 3.3.4.

The values here are all encoded in the SMTLIB2 QF_ABV logic (quantifier-free formulae over arrays and bit-vectors). The variables and registers are all represented by 8-bit and 32-bit vectors. Memory is represented as an SMTLIB array, mapping 30-bit to 32-bit words. We find that this representation results in better SMT solver performance than the obvious 32-bit to 8-bit array, since it makes the most common operations—aligned 32-bit reads and writes—simple array operations, while the rarer 8-bit reads and writes become more complex.

Some values cannot be simply encoded, for instance the heap type description and pointer validity assertions described earlier. Workarounds for this are discussed in Section 3.3.6.

3.3.3 Simple Cases and the Leaf Rule

With the problem space established, and a process available for converting variable values and path conditions to SMT expressions, the proof checker explores the proof tree. The starting assumptions are that the input variables equate as specified by the calling convention, for example $x = r_0, y = r_1$ etc.

The main objective is to show that the output variables, those taken on the arcs to the `Ret` label, are equal as specified by the calling convention. It may be assumed in proving this that the path to the `Err` label is never taken on the C side. It must also be shown that the path to `Err` is never taken on the binary side.

The `Leaf` rule instructs the proof checker to attempt to prove these final goals immediately, by converting the values of variables at `Ret` and the path conditions to `Err` into SMT values and checking that the negation of the required propositions is unsatisfiable.

3.3.4 Path Restrictions and the Restrict Rule

The conversion of variables and path conditions to SMT depends on the node of interest being reachable via some finite collection of paths. This is problematic for points which are reachable via a loop, which may be reachable after any number of loop iterations.

Consider a C function containing a single loop of this form:

```
for (i = 0; i < 4; i ++) { ...}
```

The compiler may well fully unroll this loop in the binary, since only 4 copies are needed, making the binary loop-free. The proof script must replicate this logic.

The SMT conversion process cannot describe in general the state at a node in a loop, but it can describe the 1st, 2nd or n-th visit to that node, for small values of n. The path condition for the 5th visit to the head of the loop described here can be converted to SMT, and the key observation is that this condition is always false.

The `Restrict` rule names a node and a bound n, and instructs the proof checker to check that the path condition to the nth visit to that node is unsatisfiable by SMT. The proof checker then introduces a restriction, which asserts that this node is reached less than n times. This promotes the semantic limit on the loop iterations into a syntactic limit used by the SMT conversion process, which can now handle nodes in and beyond the loop.

`Restrict` proof script nodes have a single child, which continues the proof with the new restriction in force. In the case described, the subproof may be the `Leaf` rule, which, with the restriction available, can reason about `Ret` and `Err` and finish the proof.

3.3.5 Split Induction

The `Split` rule is used to handle cases that cannot be finitely enumerated. The rule names a C split point c_sp, a binary split point b_sp, an equality predicate P and a bound n. Roughly speaking, the checker will prove by induction that for each visit to b_sp along the binary execution path, c_sp is also visited with the variable state related by P. Formally, we define c_pc_i to be the condition that c_sp is visited at least i times, b_pc_i similarly, and $|P|_i$ to be condition that the P holds on the values of the variables at the i-th visit to c_sp and b_sp respectively. Define I_i to be the property that b_pc_i implies both c_pc_i and $|P|_i$. The checker shows $\forall i > 0$. I_i by n-ary induction, that is, by proving $I_1, I_2, \ldots I_n$ directly and also that the induction hypotheses $I_i, I_{i+1}, \ldots I_{i+n-1}$ and $i > 0$ imply I_{i+n}.

Having established that the sequence of visits to b_sp is matched at c_sp, we consider three cases on the length of the sequence. If the sequence is infinite, then the binary execution and C execution are both non-terminating, and this is a valid refinement. The sequence may also contain n elements, in which case it ends with the elements $i, i+1 \ldots i+n-1$. Thirdly, the sequence may contain less than n elements. The proof script considers the latter two cases via two subproofs, which are children of the `Split` node in the proof tree. In the loop proof, new hypotheses are introduced: b_pc_{i+n-1}, $\neg b\_pc_{i+n}$, $I_i, I_{i+1}, \ldots I_{i+n-1}$. In the non-

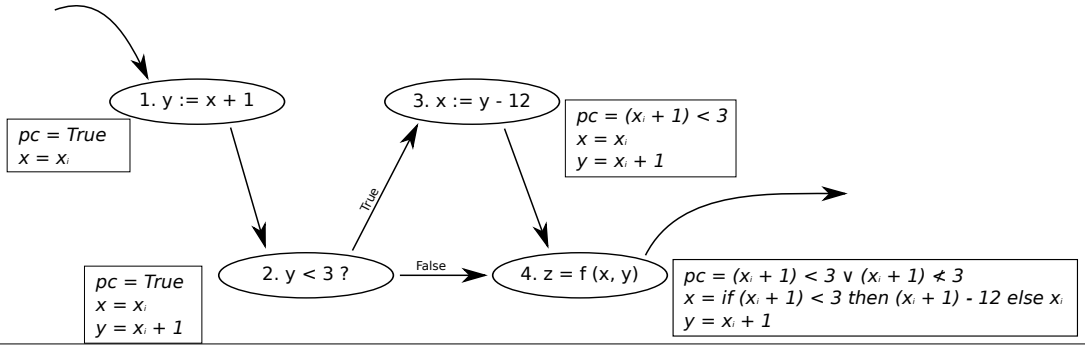

Figure 5. Example Conversion to SMT

looping case, the new hypothesis is $\neg b\_pc_n$. In each case it is expected that the subproof will begin with two `Restrict` rules which use these hypotheses to restrict the number of visits into some finite set. In the looping case, the set of possible visit counts will be of the form $\{x \mid i \leq x < i + k\}$ rather than $\{x \mid x < k\}$. This is an alternative form of the `Restrict` rule.

Some slight generalisations to this induction are needed. Firstly, the `Split` rule may define a sequence offset on either side. A C sequence offset of 2 means that we ignore the first two visits to b_sp, so b_pc_i is the condition that b_sp is visited at least $i + 2$ times, and $|P|_i$ is computed on the variable state at the second visit after the i-th visit. This may be needed to handle various optimisations which affect the initial few iterations of a loop, including a case where the binary sequence is shorter than the C sequence because some iterations have been unpacked entirely. Secondly, the predicate P may be a function not only of the variable states at the respective i-th visits, but also of the value i and the variable states at the first visit. If a C variable is incremented by 1 each iteration, it is simplest to record that it is $i - 1$ more than its first valuation.

The search process discovers the `Split` rules essentially by an exhaustive search with some minor optimisations. In practice this seems to be sufficient, although loop problems are by far the slowest problems for us to solve. In 33 of our 43 loops, the induction proof succeeds for the first candidate P for which $I_0, I_1, \ldots I_{n-1}$ hold, whereas in the remaining 10 cases the early check was mostly irrelevant and an average of 15 attempts were required to find a successful condition. The variation in these numbers is large, with the worst offending loop contributing 84 attempts, nearly half the total.

3.3.6 Assertions

Assertions are checks introduced by the C parser to ensure the standard is respected. These checks have all been handled as proof obligations in the seL4 verification, and may now be used as assumptions in this proof.

One assertion of the C standard is that no NULL pointer ever be dereferenced. The C parser produces a guard at every statement that uses a pointer which checks that the pointer is non-NULL and appropriately aligned. These guards are converted into inequalities and bit checks for the SMT solver, as are similar guards for arithmetic overflow, division by zero, etc.

Note that, for clarity, we omit these guards in our examples. There should, for instance, be a guard before the $x + 1$ calculation in Figure 5 to check that $x + 1$ does not overflow to negative.

The most involved guards relate to the strict-aliasing rule in C. The compiler is entitled to assume that no address is simultaneously in use with two different types. We adjusted the C parser to generate strong pointer validity assumptions pvalid $htd\ \tau\ p$ for every pointer p that is used with type τ when the global heap type description is htd. These assertions cannot be translated accurately into any SMT theory. Instead, each time we encounter and expression of this form, we introduce new booleans $pvalid_1$, $pvalid_2$, etc to represent them. We then translate the following key theorem:

$$\frac{\text{pvalid } htd\ \tau\ p \quad \text{pvalid } htd'\ \tau'\ p \quad \text{distinct\_types } \tau\ \tau'}{\{x \mid p \leq x < p + \text{size}(\tau)\} \cap \{x \mid p' \leq x < p' + \text{size}(\tau')\} = \{\}}$$

The SMT form of this fact is $pvalid_1 \wedge pvalid_2 \longrightarrow p + \text{size}(\tau) - 1 < p' \vee p' + \text{size}(\tau') - 1 < p$. We produce all such theorems, a possibly quadratic expansion, though the largest group of pvalid assertions on the same heap type description which we have seen in successful runs is 20.

These assertions appear in path conditions in the C function graph. The proof checker always assumes the negation of the path condition to `Err` in all its SMT checks, thus this information is always available.

	gcc -O1	gcc -O2
Instructions in Binary	11 736	12 299
Decompiled Functions	260	259
- Placeholders		3
Function Pairings	260	225
Successes	234	145
Failures	0	18
Aborted	26	62
- Machine Operations	21	13
- Nested Loops	3	2
- Machine Operations Inlined	2	47
Time Taken in Proof	59m	4h 23m

Table 1. Decompilation and Proof Results

4. Evaluation and Discussion

4.1 Results

We report on two runs of the decompilation and proof, both for gcc builds of seL4 at optimisation level 1 and 2 respectively. Table 1 shows the results. Proof timings are taken on a single core of an Intel Core 2 Duo E8400. The majority of the time taken is spent in the SMT solvers. A full decompilation run with proof certificates takes an additional 6–8 hours on modern hardware. Our implementation is based on the original decompiler implementation by Myreen et al. [17], which was not optimised for speed. Recent advances [18] may significantly improve this speed.

There are 540 functions in seL4, but far less symbols in the binary after inlining. Our proof-producing decompiler is able to process the whole binary for gcc -O1 and, at the time of writing, all

but 3 routines[2] (i.e. 98 % of the binary) at -O2. Some functions at -O2 have optimised function signatures, and thus cannot be paired via the calling convention with their C counterparts (we address this problem at the refinement stage by inlining these functions everywhere).

We produce proofs for all -O1 functions except for the 21 machine interface functions left abstract in the seL4 verification, 3 system initialisation functions involving nested loops, and 2 functions from the optional fastpath optimisation which inline machine interface operations. In the -O2 binary far more functions inline machine interface operations. There are also a number of explicit proof failures, most of which are failures to find split points, usually because of loop unrolling, loop invariant code motion and other loop structure optimisations.

We use the SMT solvers SONOLAR [23] and Z3 [7]. SONOLAR solves all of the problems we pose to it whereas Z3 times out on many of the larger problems. We suspect this is because of our heavy use of the theory of arrays (to represent memory) which SONOLAR is specifically designed to support. However Z3 supports all of the SMTLIB2 input standard, including retraction of assertions, whereas SONOLAR must be restarted for each new problem, thus we get best performance by invoking Z3 with a timeout of 2 seconds and SONOLAR if Z3 fails.

4.2 Constraints of the Approach

We aim to be general in supporting various flavours of C source, compiler and optimisation level. In practice we are limited by the restrictions of Norrish's C parser [32] we build on, our stack heuristic and the loop heuristics.

The largest single restriction imposed by the existing C parser is that it forbids taking the addresses of stack variables. The parser also forbids many uses of function pointers and all uses of variable-length argument lists [32]. In the seL4 verification this restriction simplified reasoning. As discussed in Section 3.2, it also enforces a static separation of stack accesses and heap accesses here which is needed for the decompilation process stack heuristic.

The C parser also mandates a level of conformity to the C standard which turns out to be rare in practice. For instance, duplicating a `typedef` statement, usually because of duplication of header files, is illegal in the C standard and rejected by the parser. Global variables with names beginning with underscores are also rejected. Compilers such as gcc are flexible in this regard, and various open source projects we had hoped to use as additional case studies tend to abuse this flexibility w.r.t. the standard. Both restrictions could be lifted easily in future work, since adherence to the standard becomes less important when binary verification is available.

The existing C semantics [32] as well as our graph language design fail to handle the case where the compiler has a genuine choice in data layout, such as when writing to a structure containing padding. In principle the parser and graph language could include some kind of unspecified decision node here, which would then become an existentially quantified value in the SMT problem. In practice we choose not to handle this situation, because embedded systems code often intentionally avoids padding anyway and the elimination of padding is easy to achieve on the source level.

The stack heuristic tries to discover how the binary handles the stack and tries to prove that the stack is kept separate from the heap. The exact use of the stack depends on the compiler, which means that our heuristics are fragile and at present only target recent versions of gcc. During the project, we switched gcc versions a few times. Each time the stack heuristic required some fine-tuning.

[2] These break the calling convention, i.e. gcc -O2 has performed aggressive interprocedural optimisations.

The new decompiler uses the ARM/C calling convention to make sure that the functions it extracts take input and produce output that match the input and output signatures of the corresponding C functions. This correspondence between signatures is important because it allows us to modularise the verification problem. However, when compiling with gcc -O2 we noticed that this correspondence breaks down in certain places. Sometimes gcc optimises function signatures, e.g. removing unused arguments where it can. This required special care. In the verification, we avoid these 'broken' function boundaries by inlining (in the logic) these problematic functions, i.e. such function boundaries disappear.

The loop heuristic presented in Section 3.3 depends on the existence of a split point pair, a point in the loop in the compiled binary code at which execution synchronises with a point in the C loop. There is no logical reason such a point need exist. In the case where a memory write is moved before or after an entire loop, no such point exists, and a more general formulation of problem splitting is required. This heuristic has proved sufficient at low optimisation levels, however such generalisations will be needed at higher levels of optimisation.

The loop heuristic presented here also does not handle nested loops, which do not occur in the previously verified part of seL4. To deal with nested loops, or with the compilation of loops with very complex control structure, it may be necessary to search for multiple split point pairs. This could be attempted in roughly the same manner as searching for a single split point pair, although the computational cost is likely to be high. Again, embedded systems code usually does not exhibit such complex control flow within loops, even though simple nested loops may occur. We aim to handle these in future work.

4.3 Workarounds

While we aim to support all situations as automatically as possible, we had to make a small number of changes to seL4 for the proof to work. As mentioned above, we choose not to handle the situation where the compiler is given a genuine choice in data layout. The most common cause of this in seL4 was C enumeration constants, which the C standard defines as being of a type of the compiler's choice, which must accommodate all the enumerated values. It happens that gcc on ARM picks the shortest type available, leading to padding in many structures. Furthermore the C parser guessed incorrectly that these types would be 32-bit. In principle the parser could make this flexible, but in practice in Isabelle/HOL it is more difficult to have unknown types than it is to have unknown terms.

We worked around this issue by changing the `typedef` statements used to define the types actually used in seL4 to `uint_32t` rather than any `enum` types or `uint_8t`. This eliminates all padding. There is a slight memory cost in a small number of structure types, but most structure types remain the same size.

We also adjusted a single function which loops over a short array stored in a struct in a stack variable, it being difficult for the stack heuristic to compute the bounds on the offset used in the array. Finally, we optimised a function which called a number of generic seL4 capability queries on capabilities of known type within a loop. The switch statement on the capability type appeared in every unrolling of the loop and led to an explosion in the number of possible paths causing the SMT solvers to diverge. Calling the query function specific to the capability type solved the problem.

4.4 Issues Found

We did not find any genuine compiler flaws during this analysis. The seL4 team has reported experience with compiler defects in the past and fixed all known issues by rearranging code, selecting appropriate compiler versions and disabling some compiler flags, such as -fwhole-program.

We did, however, find a number of small mismatches between C semantics and compiler. None of them were serious, but some effort was expended in removing them. These include the treatment of the strict-aliasing rule and the handling of reserved sections.

The existing proof for seL4 did not specifically address the strict-aliasing condition. It is one of the standard violations that systems code must make at some point. The C parser generated guards asserting that pointers dereferenced were aligned and not NULL, but no more. We strengthened these guards to assert that the pointers were to objects given the correct type in the heap type description to make use of that information in our SMT refinement proofs. The verification proof can be replayed essentially unchanged, since this fact was nearly always used as a step in proving the alignment conditions where they occurred anyway. Two problems remain where this strict-aliasing condition is broken.

Firstly, as a microkernel, seL4 does not use malloc and free, instead implementing its own retype mechanism. The heap type description changes during retypes, potentially invalidating the strict-aliasing condition, and there is no way to inform the compiler of this. Fortunately this will never create a problem since objects are never both accessed and retyped in the one system call.

Secondly, like standard C libraries, the kernel contains an optimised word-at-a-time memset function which is used during this retype operation. This word-wise access in memset is not compatible with the later type of these allocated objects. We had to tweak our C parser guard strengthening to explicitly omit these writes. In principle the compiler might produce unwanted code if it sufficiently inlined and reordered these functions. The fact that we prove refinement in these functions is evidence this has not happened.

Reserved sections are regions of memory which should not be adjusted by normal operations. These include the code, constant global objects and certain lookup tables generated by the compiler. With the strict stack/heap separation mandated by the C parser, this also includes the stack. The existing verification did not provide sufficient mechanisms for proving such regions were not adjusted. To include them, we once again strengthened C parser guards for the heap type description, making it clear at all locations in the C code that a given set of addresses is not covered by any type and thus unused. This set of addresses models the reserved regions.

We then produce assumptions within the SMT proof that the code and data sections created by the compiler live within this set of reserved addresses, as does the stack. While we can prove this condition is maintained, we have to assume it for the initial state. It would be desirable to make this assumption into a proof in future work, which would essentially extend the verification to the binary image as loaded in memory, as opposed to the binary image in the ELF file, i.e. it would further remove loading and in-memory relocation from the trusted computing base.

5. Related Work

Pnueli et al [24] proposed translation validation as a pragmatic alternative to compiler verification. Over a decade of competition between these two approaches has yielded a collection of translation validation experiments [4, 9, 12, 19, 26, 30, 35, 36] as well as a few verified compilers [5, 13]. This distinction is frequently blurred: some phases of the CompCert C compiler use translation validation whereas others are directly verified.

Our approach is yet another variation within the space of possible translation validations. Like Tristan et al [30], we do not make use of any hints from the compiler. Like Ryabtsev [26], we target the end-to-end transformation rather than any internal compiler step. Furthermore we do not accept any failures or false positives in the process. However, unlike other authors, we are prepared to make small adjustments to our source code if necessary.

What differentiates this approach is that it is strongly grounded in existing semantics at both ends, rather than the compiler's view of its input and output. At the binary level we connect to the extensively validated Cambridge ARM semantics, and at the source level we connect to the verifier's view of the C language which has been validated as useful through the existing seL4 proofs.

The Verisoft project [1] also uses a verified source to binary conversion to produce a verified binary. The project developed a verified compiler for a Pascal-like language with C-like syntax which shared its semantic framework with the verification environment. While the project clearly showed that end-to-end theorems to the binary level are feasible, practical considerations such as performance were not goals of the project.

The CompCert [13] verified C compiler could also be used to transport source-level proofs to the binary, by building a program logic on the semantics associated with the compiler's specification [2]. In our case those semantics are presented differently to those assumed in seL4's verification, and in an incompatible logic.

An alternative approach to binary verification is to prove results directly on the binary semantics with interactive tools. For instance, Bevier's KIT [3] was the first operating system to be completely verified at the assembly level. However, it measured only a few hundred lines of assembly in total. More recently, Ni et al [20] verified modern context switching code using the XCAP x86 model. Chlipala [6] presents a more sophisticated suite of tools for automating most of such reasoning.

Yang and Hawblitzel [33] used binary verification on a minimal type safe language runtime, including garbage collection, to implement an OS kernel. The verification establishes type safety for user-level programs, but forces all applications into the same language framework. Our verification achieves full functional correctness of the kernel instead, and seL4 uses hardware mechanisms to enforce isolation for arbitrary application programs.

6. Conclusion

We have extended the existing formal functional correctness proof of the seL4 microkernel from the C source code down to the binary level. This means, the seL4 binary conforms to its high level correctness properties stated in Isabelle/HOL. The C source, its semantics, and the compiler need no longer be trusted.

Our approach validates the output of compiler and linker by proving refinement between the formal semantics of a program on the C source level and its formal semantics on the binary level. Our refinement theorem composes with further refinement stacks on top of the source code such as in seL4, and transports Hoare-logic properties proved on the source code down to the binary.

We achieved this by building on an existing well-established C semantics and strongly validated ARM semantics and decompilation framework, extending both frameworks and translating them to a common intermediate format particularly amenable to modern high-performance SMT solvers.

While our main target was the seL4 microkernel, we believe that this approach in principle generalises to other C verifications, to other compilers, and even to other similar programming languages. We think that the limitations of the C parser front-end and the prover back-end, such as nested loops, can be overcome in future work to result in a tool chain for fully automatically extending formal C verification down to the binary level. This removes compilers from the trusted computing base of high-assurance systems, while still enabling the use of off-the-shelf compilation tool chains.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence

program. The second author is funded by the Royal Society, UK. This work was partially supported by EPSRC Research Grant EP/G007411/1.

References

[1] E. Alkassar, W. Paul, A. Starostin, and A. Tsyban. Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In P. O'Hearn, G. T. Leavens, and S. Rajamani, editors, *VSTTE 2010*, volume 6217 of *LNCS*, pages 71–85, Edinburgh, UK, Aug 2010. Springer.

[2] A. W. Appel. Verified software toolchain—(invited talk). In G. Barthe, editor, *Proc. 20th ESOP*, volume 6602 of *LNCS*, pages 1–17, Saarbrücken, Germany, Mar. 2011. Springer.

[3] W. R. Bevier. Kit: A study in operating system verification. *IEEE Trans. Softw. Engin.*, 15(11):1382–1396, 1989.

[4] J. O. Blech, I. Schaefer, and A. Poetzsch-Heffter. Translation validation of system abstractions. In *Proc. 7th Int. Conf. on Runtime verification*, RV'07, pages 139–150, Vancover, Canada, 2007. Springer.

[5] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In J. Ferrante and K. S. McKinley, editors, *Proc. PLDI'07*, pages 54–65. ACM, 2007.

[6] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proc. 32nd PLDI*, pages 234–245, San Jose, California, USA, 2011. ACM.

[7] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *14th TACAS*, volume 4963 of *LNCS*, pages 337–340, Budapest, Hungary, Mar. 2008. Springer.

[8] A. Fox and M. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In M. Kaufmann and L. C. Paulson, editors, *1st Int. Conf. Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 243–258, Edinburgh, UK, July 2010. Springer.

[9] B. Goldberg, L. D. Zuck, and C. W. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. Proc 3rd Int. Workshop on Compiler Optimization Meets Compiler Verification (COCV '04). *Electr. Notes Theor. Comput. Sci.*, 132(1):53–71, 2005.

[10] ISO/IEC. Programming languages — C. Technical Report 9899:TC2, ISO/IEC JTC1/SC22/WG14, May 2005.

[11] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd SOSP*, pages 207–220, Big Sky, MT, USA, 2009. ACM.

[12] S. Kundu, S. Lerner, and R. K. Gupta. Translation validation of high-level synthesis. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29(4):566–579, Apr. 2010.

[13] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *Proc. 33rd POPL*, pages 42–54. ACM, 2006.

[14] X. Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43(4):363–446, 2009.

[15] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symp. Security & Privacy*, Oakland, CA, May 2013.

[16] M. O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2009.

[17] M. O. Myreen, M. J. C. Gordon, and K. Slind. Machine-code verification for multiple architectures - an application of decompilation into logic. In A. Cimatti and R. B. Jones, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–8. IEEE, 2008.

[18] M. O. Myreen, M. J. C. Gordon, and K. Slind. Decompilation into logic — improved. In G. Cabodi and S. Singh, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 78–81. IEEE, 2012.

[19] G. C. Necula. Translation validation for an optimizing compiler. In *Proc. PLDI'00*, pages 83–94, Vancouver, BC, Canada, 2000. ACM.

[20] Z. Ni, D. Yu, and Z. Shao. Using XCAP to certify realistic systems code: machine context management. In *Proc. 20th TPHOLs*, volume 4732 of *LNCS*, pages 189–206. Springer, 2007.

[21] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[22] Open Kernel Labs. seL4 research and evaluation download. `http://ertos.nicta.com.au/software/seL4/`, 2011.

[23] J. Peleska, E. Vorobev, and F. Lapschies. Automated test case generation with SMT-solving and abstract interpretation. In *Proc 3rd Int. Conf. NASA Formal methods*, NFM'11, pages 298–312, Pasadena, CA, 2011. Springer.

[24] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *Proc. 4th TACAS*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.

[25] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74. IEEE Computer Society, 2002.

[26] M. Ryabtsev and O. Strichman. Translation validation: From Simulink to C. In *Proc. 21st Int. Conf. on Computer Aided Verification*, CAV '09, pages 696–701, Grenoble, France, 2009. Springer.

[27] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[28] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. In M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *2nd Int. Conf. on Interactive Theorem Proving*, volume 6898 of *LNCS*, pages 325–340, Nijmegen, The Netherlands, Aug. 2011. Springer.

[29] K. Slind and M. Norrish. A brief overview of HOL4. In *20th Int. Conf. on Theorem Proving in Higher Order Logics*, pages 28–32, Montreal, Canada, Aug. 2008.

[30] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for llvm. In *Proc. 32nd PLDI*, pages 295–305, San Jose, CA, USA, 2011. ACM.

[31] H. Tuch. Formal verification of C systems code: Structured types, separation logic and theorem proving. *J. Automated Reasoning: Special Issue on OS Verification*, 42(2–4):125–187, Apr. 2009.

[32] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proc. 34th POPL*, pages 97–108, Nice, France, Jan. 2007. ACM.

[33] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proc. PLDI'10*, pages 99–110, Toronto, Canada, 2010. ACM.

[34] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In M. W. Hall and D. A. Padua, editors, *Proc. 32nd PLDI*, pages 283–294, San Jose, CA, USA, June 2011. ACM.

[35] L. D. Zuck, A. Pnueli, Y. Fang, B. Goldberg, and Y. Hu. Translation and run-time validation of optimized code. Runtime Verification 2002 (RV'02). *Electr. Notes Theor. Comput. Sci.*, 70(4):179–200, 2002.

[36] L. D. Zuck, A. Pnueli, and B. Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *J. UCS*, 9(3):223–247, 2003.